Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de la Telecomunicación

Sistema centralizado para el control de integridad y mitigación de intrusiones en puestos de trabajo

Autor: Verónica Escarlata Berenguer Garrido
Tutor: Pablo Nebrera Herrera

Dep. Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2016

Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías de la Telecomunicación

Sistema centralizado para el control de integridad y mitigación de intrusiones en puestos de trabajo

Autor:
Verónica Escarlata Berenguer Garrido

Tutor:
Pablo Nebrera Herrera
Profesor titular

Dep. Telemática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2016

Trabajo Fin de Grado: Sistema centralizado para el control de integridad y mitigación de intrusiones en puestos de trabajo

Autor: Verónica Escarlata Berenguer Garrido

Tutor: Pablo Nebrera Herrera

El tribunal nombrado para juzgar el Trabajo arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2016

El Secretario del Tribunal

A mi familia

A Adrián

A Chiquitín

Agradecimientos

Hace 6 años que todo comenzó. Hoy estoy cerrando un ciclo en mi vida con estas palabras, ciclo del que han formado parte muchas personas. Sin algunas de ellas posiblemente no estaría aquí en estos momentos, y sin otras no habría sido lo mismo.

Empiezo dando las gracias a mi familia en general, tanto directa como política, pero en especial a mis padres y hermana. El apoyo que me han brindado durante esta etapa no tiene nombre. Todavía no he escuchado ni una sola queja o recriminación por esos suspensos que me han estado acompañando, todo lo contrario, sólo he escuchado ánimos. Y esas alegrías por los aprobados no pueden describirse con palabras. Simplemente, gracias.

Sigo con mi prima, la persona que hizo que me "entrara el gusanillo" por la ingeniería. Sin ella, quién sabe si hubiera acabado finalmente donde estoy. Gracias Merchistere por todos tus consejos.

Continuo con esos amigos y compañeros de teleco con los que he ido forjando una gran amistad. Creo que los ingenieros tenemos una segunda familia, una familia que tenemos la suerte de elegir para que formen parte de nuestra vida, con los que convivimos alrededor de 10 horas al día entre asignaturas, prácticas y trabajos; los que nos comprenden como nadie porque han vivido en sus carnes lo mismo que uno, con los que desahogarnos tras las caídas y alegrarnos con las victorias. Porque sin vosotros esta etapa no hubiera sido la misma, y si la volviera a vivir no la querría sin vosotros. Gracias Azahara, Lucía, Javi, Gema, Juanan, Alberto, Postigo...

A parte de las nuevas amistades que surgen, nunca hay que olvidar con las que una crece. Prácticamente toda mi vida a vuestro lado, 18 años van ya y lo que quedan. Gracias Carmen, Juanmi, Alba y Minerva por todos esos buenos momentos vividos y por hacer que esta amistad no tenga fecha de caducidad.

Lo que tengo muy claro es que, sin su confianza, este proyecto no hubiera estado en mis manos. Gracias Pablo por haberme dado la oportunidad de haber sido un miembro más en Redborder y, no sólo eso, sino de aprender y haber "roto el hielo" con el mundo laboral. También agradecer a todo el equipo de desarrollo, web, sistemas, etc. por haberme ayudado en todo lo que he necesitado. En particular me gustaría dar las gracias a Carlos Jiménez por haberme ofrecido este reto, preocuparse por enseñarme cosas nuevas y guiarme en mi paso por la empresa.

Para el final he querido dejar a la persona que ha ido formando parte de mi día a día durante estos últimos años. Has sido un pilar fundamental en estos años, sobre todo los que abarcan este ciclo. He tenido la suerte de tenerte como amigo, pareja y compañero. Nos hemos apoyado mutuamente en todo, hemos festejado nuestros logros y hemos podido desahogarnos en nuestros peores momentos. El año pasado fue muy duro para mí en lo personal y tú siempre estuviste ahí, haciendo que naciera en mí una fuerza que nunca pensé que podría llegar a tener. Tú me enseñaste a razonar y que la vida no es de color de rosa, que hay que trabajarla y, con esfuerzo, todo se consigue. Gracias, Adrián. Hemos comenzado y acabado juntos, sin duda la anécdota más bonita que siempre voy a recordar de mi paso por esta etapa.

Verónica Escarlata Berenguer Garrido

Sevilla, 2016

Resumen

Ha habido, hay, y siempre habrán nuevas amenazas que nos acechen, por lo que hay que ir renovando las medidas de detección y protección de los dispositivos que hacen uso de Internet para estar actualizados y hacer frente a las nuevas formas de malware que puedan aparecer.

Como base a ello nace este proyecto, cuyo objeto es anticiparse a la amenaza ante la aparición de comportamientos inusuales en los equipos finales de los usuarios. Para ello se apuesta por nuevas tecnologías como Google Rapid Response (GRR) para la monitorización de estos equipos. Gracias a ello podremos obtener un análisis periódico de éstos y, así, poder recopilar los estados de los sistemas finales. Con esta información se pueden comparar dichos estados y alertar de una posible intrusión en caso de hallar un cambio significante. Al ser importante la contención de los equipos que se encuentren en cuarentena, mientras se comprueba si una amenaza es real se cortará toda comunicación entrante y saliente con el exterior hasta que el peligro haya pasado.

Esta herramienta está más orientada al ámbito empresarial, ya que se aprovechan más los recursos al analizar periódica y simultáneamente todos los equipos que en él se encuentran en busca de amenazas, realizando las medidas oportunas en caso de hallazgo.

Abstract

There has been, is, and will always be new threats that lurk us, so we must be renewing detection measures and protection devices that make use of the Internet to be updated and to address new forms of malware that may appear .

As this basis born this project, which aims to anticipate the threat to the appearance of unusual behavior in the end users' computers. For it is committed to new technologies like Google Rapid Response (GRR) for monitoring these teams. As a result we can obtain a periodic analysis of these and thus to collect status of end systems. With this information you can compare these states and warn of a possible intrusion in case of finding a significant change. As important containment equipment on quarantined while it is checked whether a threat is real all incoming and outgoing communication with the outside is cut until the danger has passed.

This tool is more oriented to the business because resources are more used to their regular and simultaneously analyze all the teams that are in it for threats, making the appropriate measures in case of discovery.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

Notación

TFG	Trabajo Fin de Grado
SO	Sistema Operativo
GRR	Google Rapid Response
ID	Identificador de petición GRR
BBDD	Base de datos
CSR	Certificate Signing Request
EndPoint	Equipo final
CSV^{G}	CSV Gigante
CSV^{GN}	CSV Gigante de Netstat
CSV^{GP}	CSV Gigante de ListProcesses
CSV^{I}	CSV Individual
CSV^{IN}	CSV Individual de Netstat
CSV^{IP}	CSV Individual de ListProcesses
CSV^{R}	CSV Resumido
CSV^{RN}	CSV Resumido de Netstat
CSV^{RP}	CSV Resumido de ListProcesses
p.e	Por ejemplo
VM	Máquina Virtual
FW	Firewall
G.I.T.T	Grado en Ingeniería de las Tecnologías de la Telecomunicación

1. INTRODUCCIÓN

"La vida no es fácil, para ninguno de nosotros. Pero … ¡Qué importa! Hay que perseverar y, sobre todo, tener confianza en uno mismo. Hay que sentirse dotado para realizar alguna cosa y que esa cosa hay que alcanzarla, cueste lo que cueste".

- Marie Curie -

No es raro encontrar constantemente noticias relacionadas con malware y cómo éste puede afectar en nuestra navegación por la red. Además, encontramos nuevos objetos informatizados en nuestro día a día que podrían verse amenazados. El internet de las cosas ha convertido nuestro entorno en una red de objetos conectados a la red que pueden ser vulnerables a ataques malintencionados.

Además de contar con un número considerable de amenaza malware también existen herramientas que trabajan para impedir que nos puedan afectar. La pregunta es: ¿Es posible prevenir el malware antes de que infecte nuestros equipos? Sí, es posible. Si analizamos los cambios de estado de los equipos cliente y observamos anomalías en su comportamiento, seremos capaces de actuar antes de que el malware se haya propagado en nuestra red. Lo ideal sería, asimismo, poder poner en cuarentena los equipos afectados mientras se corrobora la posible intrusión.

Partimos de estas dos ideas para la creación de una nueva herramienta que se sume a la lucha contra el malware, desarrollada para el proyecto redBorder Malware, la cual tiene como objeto la monitorización de equipos finales para llevar a cabo sus diversas labores de prevención.

Veremos las diferentes partes de esta herramienta y cómo funciona en capítulos posteriores.

1.1 Seguridad en la red

Podemos observar como todo nuestro día a día gira entorno a aparatos informatizados: smartphones, tablets, ordenadores, TV, etc. Con ello nacen a la misma vez nuevas aplicaciones, hardwares, SO, etc que, como todo sistema innovador, es más débil en sus comienzos. Los ciberdelincuentes se aprovechan de esta vulnerabilidad (incluso de las personas que hacen uso de

ellos) para obtener a cambio un beneficio, por lo que es importante estar protegidos.

Definición (Seguridad informática): "*La seguridad informática consiste en la implantación de un conjunto de medidas técnicas destinadas a preservar la confidencialidad, la integridad y la disponibilidad de la información, pudiendo, además, abarcar otras propiedades, como la autenticidad, la responsabilidad, la fiabilidad y el no repudio.*" (ISO/IEC 27001:2005 Information technology — Security techniques — Information security management systems – Requirements)

La necesidad de proteger nuestros datos frente a intrusos surge de la amenaza de éstos por obtener información ajena. El mundo tecnológico está evolucionando de manera prominente en estos últimos años, a la misma vez que las técnicas de ataque, por lo que no hay que dejar en segundo plano la ciberseguridad. Hay que innovar y crear barreras de contraataque efectivas para enfrentar todo tipo de malware.

1.1.1 Aspectos de la ciberseguridad

Dentro del estudio de la ciberseguridad podemos encontrar tres factores fundamentales que determinarán la fiabilidad de un sistema. Estamos hablando de la confidencialidad, integridad y disponibilidad.

- *Confidencialidad:* Básicamente se refiere a que sólo aquellos usuarios que estén autorizados tengan acceso a la información.
- *Integridad:* La información transmitida entre un emisor y un receptor no debe ser corrompida por un tercero.
- *Disponibilidad:* Aquí la información debe ser accesible en todo momento por los usuarios autorizados.

Figura 1.5 Propiedades de la ciberseguridad

Hay que conseguir un equilibrio entre éstos para poder brindar seguridad, pudiendo priorizarlos según las preferencias del usuario.

1.2 redBorder

redBorder es una plataforma orientada a la ciberseguridad cuya función es la monitorización y análisis del tráfico para proteger a los usuarios de toda amenaza malware.

Uno de los proyectos más ambiciosos en los que se ha embarcado ha sido redBorder Malware, destinado a la ciberseguridad, en el que se analizan los datos que transcurren por la red para

evitar acceso de malware. La herramienta realizada por el autor del trabajo fin de grado ha sido desarrollada para ser incluida en dicho proyecto.

Para lograr su objetivo, redBorder Malware se apoya en dos herramientas: Managers y Sensores. Los primeros se encargan de procesar los datos recolectados por los segundos en la red del cliente. A través de la interfaz web se puede ver de manera más intuitiva para el usuario todo este proceso y cuál ha sido la evaluación final tomada para unos datos dados.

redBorder Malware hace uso de una serie de servicios para su correcto funcionamiento. En el siguiente capítulo, y a lo largo de la memoria, serán explicados aquellos que han sido necesitados por el alumno para llevar a cabo el TFG.

1.3 Motivación

Las ganas de contribuir en la defensa de ataques malintencionados se suman con la oportunidad que le ha brindado redBorder al autor del TFG[1] de participar en un proyecto de tales características como es redBorder Malware. Como resultado de ello nace este trabajo, cuyo objetivo es crear una herramienta de detección malware en puestos de trabajo.

La dificultad previa al desarrollo estaba, en gran parte, en el estudio y comprensión de la herramienta desarrollada por Google, denominada Google Rapid Response[2] o GRR. A esto se le añadía el handicap del lenguaje de programación usado para su funcionamiento. Nos referimos a Python. Lo denominamos como obstáculo debido al poco conocimiento que tenía el alumno de él.

Otro de los retos a los que el alumno ha tenido que enfrentarse ha sido el estudio del avanzado estado del proyecto redBorder Malware, cuya versión en el instante de la incorporación era la 0.8. Era necesario entender la estructura y funcionamiento del mismo para poder darle a su trabajo el enfoque apropiado, a la vez que poder que integrarse en el equipo de trabajo y, así, proponer nuevas ideas y encontrar soluciones a problemas surgidos.

No sólo por la atracción de la temática del TFG es por lo que el alumno ha tomado las riendas del mismo, sino también por el aprendizaje que iba a adquirir al enfrentar los obstáculos comentados.

1.4 Objetivos

Hoy en día es necesario que las empresas dedicadas a la ciberseguridad estén informadas tecnológicamente hablando de las innovadoras herramientas anti-malware que salen al mercado. Gracias a ello podrán conocer la oferta y demanda para poder competir y estar a la altura de las mismas.

Debido al descubrimiento de la herramienta desarrollada por Google, GRR, es cómo surgió la idea de este proyecto. Su capacidad de recolección de datos y la gran cantidad de equipos que puede llegar a soportar simultáneamente la convierte en un fuerte apoyo de combate malware.

[1] Con estas siglas nos referiremos al Trabajo Fin de Grado del alumno a lo largo de la memoria.
[2] A lo largo de la memoria nos referiremos a este software como GRR

La finalidad de su uso era aprovechar su función de monitorización para el envío masivo de peticiones y recepción de sus correspondientes resultados. A partir de aquí es donde entra en juego el alumno del proyecto. Uno de los objetivos del mismo es que estos informes fueran procesados a través de un software que permita comparar en diferentes instantes de tiempo los distintos resultados. Si se viera una significante alteración podría implicar la presencia de código maligno dentro del sistema final. Un analista sería el encargado de realizar un estudio posterior del equipo para corroborar esta alerta.

Otro de los propósitos es inhabilitar toda conexión entrante y saliente de los equipos que están siendo analizados. Para ello está planeada la contención mediante el firewall de Windows de éstos mientras perdure su análisis.

En el tercer capítulo veremos en más detalle GRR para comprender su funcionamiento y cualidades, ya el alumno se apoyará en esta herramienta para llevar a cabo el TFG.

1.5 Metodología de trabajo

En todo proyecto están implicadas dos partes: el cliente[3] y el grupo de trabajo[4].

Con el primero de ellos era necesario un feedback[5] periódico para discutir las diversas peticiones y cambios del proyecto, al igual que conocer sus opiniones respecto al avance del mismo. Se han usado varios medios para llevar a cabo dicha comunicación. Entre ellos destacamos Gmail, Basecamp y GoToMeeting. La segunda de ellas es una aplicación con la que gestionar proyectos, a la vez que mantener contacto con los clientes involucrados; mientras que la tercera y última es aquella con la cual se pueden realizar videoconferencias online en tiempo real, siendo ésta la preferida para mostrar al cliente el funcionamiento y pruebas del proyecto en varias sesiones. Los escasos encuentros presenciales son debidos a la distancia que separa las dos sedes implicadas: una ubicada en Sevilla y la otra en México.

Las diversas reuniones con el grupo de trabajo eran realizadas presencialmente en su mayoría para valorar las distintas mejoras y soluciones frente a obstáculos encontrados. Las aplicaciones más frecuentemente usadas para la comunicación entre los miembros del equipo son Slack, herramienta en forma de chat con diferentes opciones, como el envío de ficheros y creación de grupos; y Redmine, con la que se asignaban las diversas tareas y se llevaba constancia del timeline del proyecto.

[3] El cliente del proyecto redBorder Malware es Produban, empresa tecnológica del Grupo Santander con sedes en varios países. El responsable del mismo se encuentra en México

[4] redBorder dispone de dos sedes, ambas en España: Madrid y Sevilla. El alumno del TFG ha realizado el mismo en la segunda de ellas.

[5] Con feedback nos referimos al hecho de tener retroalimentación por parte del cliente de los temas planteados, y viceversa.

2.RECURSOS

"La única manera de hacer un gran trabajo es amar lo que hace. Si no ha encontrado todavía algo que ame, siga buscando. No se conforme. Al igual que los asuntos del corazón, sabrá cuando lo encuentre"

- Steve Jobs -

Para poder llevar a cabo este proyecto han sido necesarios una serie de recursos tanto software como hardware. Éstos son indispensables si queremos reproducir el escenario en alguna ocasión, sin los cuáles no obtendremos los resultados esperados. A continuación, haremos mención de ellos.

2.1 Recursos Hardware

2.1.1 Desarrollo del software

El único recurso hardware que ha sido necesario para la implementación de este proyecto es un Intel NUC al que se le ha incluido el SO[6] Ubuntu 12.04, un mini ordenador que se enlaza a cualquier pantalla de ordenador de sobremesa y realiza las mismas funciones que éste, aunque con menos potencia y alcance.

2.1.2 Equipamiento de sistemas finales

En caso de que se desee elaborar un entorno para hacer uso de la herramienta creada en este proyecto, serán necesarios una serie de equipos con SO Windows, Ubuntu o iOS. No hay un mínimo establecido, puede utilizar el número de equipos que considere oportuno. El papel que desempeñarán será el de equipos analizados.

Dentro del proyecto han sido utilizados[7] para corroborar la perfecta ejecución de la herramienta. Debido a la cantidad necesaria para poder realizar en ellos las pruebas[8] se han tenido que hacer uso de máquinas virtuales en lugar de físicas.

[6] Sistema Operativo
[7] Con S.O Windows 7.
[8] Han sido usados entre 20 y 30 equipos.

2.2 Recursos Software

Los recursos software de los que hemos dispuesto para el correcto funcionamiento del proyecto son los que describimos a continuación. No nos meteremos en profundidad en la explicación de cada uno de ellos debido a que en capítulos posteriores entenderemos mejor sus papeles en el proyecto.

2.2.1 Google Rapid Response (GRR)

Sistema de monitorización de equipos finales. Es necesario para realizar un informe de los mismos y, así, poder evaluar la presencia de ataque malware. En el siguiente capítulo explicaremos con más exactitud su funcionamiento y rol en este proyecto.

Figura 2.1 Logo del sistema de monitorización GRR [1]

2.2.1.1 Requisitos

Los requisitos[9] [2] que se deben cumplir para la instalación del agente servidor de GRR son:

- Máquina física o virtual de 64 bits.
- SO Ubuntu Server 14.04.
- Es recomendable una capacidad mayor de 1GB de RAM.

Para la instalación del agente cliente de GRR en equipos finales tenemos los siguientes requerimientos:

- Máquinas físicas o virtuales con SO Windows, Linux y OSX.

2.2.2 Máquinas virtuales

Una máquina virtual es un software que imita las funcionalidades y comportamientos de equipos físicos. Gracias a ella podremos obtener distintos SO en un mismo sistema sin realizar las correspondientes particiones en el disco duro.

Este tipo de software ha sido utilizado en el TFG del alumno para crear sistemas finales tanto de 32 como de 64 bits y con SO Windows 7. Éstos eran usados para instalar los agentes clientes de GRR.

Además de lo anterior, también ha sido necesario crear mediante este medio el agente servidor GRR.

[9] La información acerca de los requisitos necesarios, tanto para el agente cliente como el servidor de GRR, ha sido obtenida de la página de github de GRR. El enlace web al documento está especificado en el apartado de Referencias.

2.2.3 redBorder Malware

Es una plataforma de seguridad, gestión y monitorización para garantizar detección y protección frente a amenaza malware. Está compuesta por una serie de sensores ubicados en diferentes partes de la red y gestionados desde el Manager[10].

Este producto es objeto del proyecto redBorder Malware, y es en él donde vamos a integrar el software desarrollado por el alumno del TFG, concretamente en el Manager de redBorder.

Figura 2.2 Logo de redBorder [3]

2.2.4 Apache kafka

Es una cola de mensajes en la que una serie de productores publican eventos, identificados mediante un tema o topic, a los suscriptores de los mismos. Éstos solo consumirán aquellos mensajes de los topics a los que estén suscritos.

Dentro de este proyecto hará falta que recurramos a este sistema de paso de mensajes, incorporado en el Manager de redBorder, para informar cuando se produzca un cambio de estado en alguno de los sistemas finales que estén monitorizados por GRR.

El papel que desempeña esta herramienta a lo largo del TFG será explicada en el capítulo 4.

Figura 6.3 Logo de Apache kafka [4]

2.2.5 Aerospike

Aerospike es una base de datos NoSQL (no relacional), distribuida, con modelo de datos clave – valor, de código abierto y escalable. Está especialmente diseñada para tener una gran capacidad de escalabilidad, manteniendo un breve tiempo de respuesta y una total disponibilidad.

Podemos encontrar esta herramienta, al igual que la anterior, dentro del Manager de redBorder. Será usada en el TFG del alumno para detectar si ha habido un cambio de estado en alguno de los sistemas finales que estén monitorizados por GRR.

El papel que desempeña esta herramienta a lo largo del TFG será explicada en el cuarto capítulo de la memoria.

[10] Con este término denominaremos a lo largo de la memoria al Manager de redBorder

Figura 2.4 Logo de Aerospike [5]

2.2.6 Amazon S3

Es un servicio de almacenaje de ficheros en la cloud.

Esta herramienta será utilizada por el alumno del TFG para acumular en ella sólo los nuevos informes obtenidos por GRR de los equipos finales.

El papel que desempeña esta herramienta a lo largo del TFG será explicada, al igual que las anteriores, en el capítulo 4.

Figura 2.5 Logo de Amazon S3 [6]

2.2.7 Python

Python es el lenguaje de programación elegido por el alumno del TFG para llevar a cabo el desarrollo del mismo[11].

Destaca por ser interpretado, multiplataforma y orientado a objetos; además de imperar en él una serie de reglas estilísticas para garantizar la legibilidad al usuario. Estas han sido algunas de las razones que han llevado al alumno a optar por este lenguaje. Si a esto le sumamos que GRR está en su totalidad desarrollado en él lo convierte en la herramienta de desarrollo ideal para el proyecto.

Figura 2.6 Logo de Python [7]

[11] La versión de Python usada para el proyecto ha sido la 2.6.

2.3 Entornos de desarrollo

2.3.1 Sublime Text

Para la codificación de este proyecto el aliado elegido por el alumno ha sido el editor de texto Sublime Text. La comodidad de navegación por pestañas y sus características de formateo la convierten en una herramienta muy útil y sencilla para un programador de python.

Figura 2.7 Logo de Sublime Text

3.GOOGLE RAPID RESPONSE (GRR)

"La función de un buen software es hacer que lo complejo aparente ser simple".

- Grady Booch -

3.1 Introducción a GRR

Google Rapid Response[12], comúnmente denominado GRR, es un sistema de monitorización y análisis forense que da soporte a equipos finales, dando respuesta a cualquier incidencia ocurrida en éstos.

Esta herramienta se divide en dos secciones: una vinculada a la parte del cliente y otra a la del servidor. El primero de ellos es desplegado en forma de agente en equipos finales, interaccionando mediante acceso remoto con el servidor para proporcionarle la información demandada por éste.

A continuación, vamos a ver las características de este sistema:

- Multiplataforma (Windows, Linux y OSX)
- Distribuido
- Escalable (puede alcanzar más de 100K equipos)
- Programable a través de scripts
- De código abierto[13]
- Comunicación cliente – servidor segura, diseñada para el despliegue en Internet a través de HTTP

Desde el servidor se pueden realizar una serie de solicitudes, comentadas a continuación:

- Importación del sistema de ficheros del cliente
- Potente capacidad de búsqueda y descarga de ficheros y claves del registro de Windows
- Análisis de la memoria del cliente

[12] A lo largo de la memoria nos referiremos a este software como GRR. Se ha hecho uso de esta referencia para la recolección de información de GRR [8].
[13] También denominado Open Source

- Información de conexiones de red
- Recolección pruebas forenses (Artefactos Forenses)
- Recolección de datos y capacidad de analizarlos
- Monitorización del equipo cliente

La idea básica de este sistema es que desde el servidor se envían peticiones hacia los clientes solicitando cierto tipo de información[14] para proceder al análisis de los equipos finales. Ante una incidencia ocurrida en éstos, un analista podría ser el que, desde el servidor, pida esa información para estudiarla y asegurar algún indicio de amenaza.

3.2 Operaciones

El envío de solicitudes que se realizan desde el servidor al cliente se les conoce como Flows o Hunts. Veamos a continuación qué significado tienen.

3.2.1 Flows

A modo de resumen, los flows son un conjunto de peticiones lanzadas desde el servidor GRR hacia un[15] cliente para obtener cierta información acerca de éste. Una vez que la obtengamos podremos analizarlo.

La siguiente imagen muestra el envío de los múltiples mensajes remitidos entre el cliente y el servidor para una petición cualquiera.

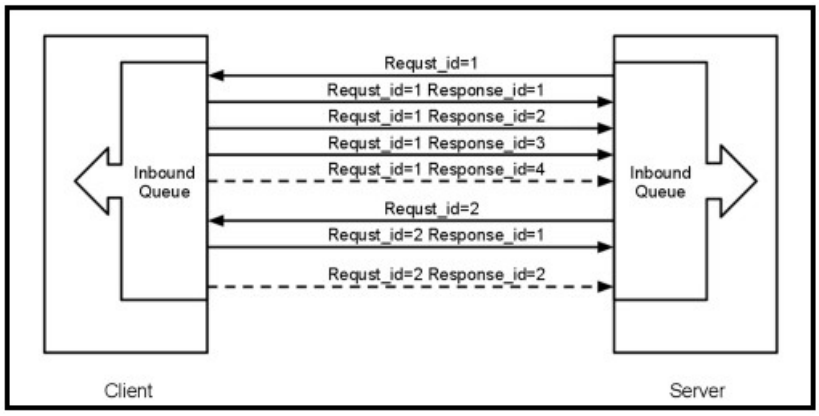

Figura 3.1 Envío de mensajes entre el cliente y el servidor GRR para un flow determinado [9]

Expliquemos con más detalle el uso de cada uno de ellos y de sus parámetros.

3.2.1.1 Request_id

Los flows se vinculan a un identificador[16] aleatorio, denominado Request_id. Éste se creará para cada flow que sea lanzado. Todas las respuestas del cliente a un flow concreto se envían al

[14] Con " cierto tipo de información" nos referimos a la lista de solicitudes que acabamos de mencionar.
[15] Sólo a uno
[16] A lo largo de la memoria también usaremos el término ID para referirnos a identificador.

servidor usando su identificador. Como podemos ver en la imagen anterior, para una misma solicitud se generan varios mensajes de respuesta, todos ellos con el mismo ID.

3.2.1.2 Response_id

Nos encontramos ahora con el parámetro Response_id que, al contrario del anterior, sí modifica su valor, incrementándose secuencialmente para cada mensaje de respuesta. Esto es debido a que se refiere al orden en que se transmiten dichos mensajes.

3.2.1.3 Mensaje STATUS

Observemos como el último mensaje de ese conjunto de respuestas[17] está representado en la imagen diferente a los anteriores. Esto se debe a que éste no transmite el mismo tipo de contenido que los demás, ya que indica que todas las respuestas han sido enviadas a la solicitud del servidor. Además, también avisa si ha habido algún error o todo se ha realizado correctamente. Este tipo de mensaje se llama STATUS.

Una vez que conocemos el funcionamiento de un flow y sus parámetros podemos incluir una definición más técnica de lo que es.

Definición (Flow): Un flow es, ni más ni menos, una máquina de estados, en la que, en cada estado por el que pasa, envía una respuesta al servidor hasta completar la solicitud que éste le había pedido.

3.2.1.4 Equipo final no disponible

En caso de que el servidor envíe un flow a un cliente y éste se encuentre apagado la petición se suspende y se serializa en el disco hasta que vuelva a estar disponible.

Los flows están en memoria sólo mientras se ejecutan, donde van recibiendo las respuestas del cliente. Si el equipo final no está en funcionamiento el flow se queda serializado en el disco y no se usa memoria. Así, por ejemplo, si un cliente desaparece a la mitad de la copia de un archivo ni se pierde la información que se estaba transfiriendo ni se gasta memoria, y, además, cuando éste vuelva se reanudan las operaciones que se estaban realizando antes de su indisponibilidad. Los flows se quedan esperando en el estado en el que estaba antes de su interrupción.

3.2.2 Hunts

Definición (Hunt): Los hunts son iguales que los flows, pero en lugar de lanzar las peticiones a un solo cliente lo hacen a varios a la vez.

Los equipos clientes se pueden agrupar en el servidor identificados por una etiqueta[18], por lo que un hunt se puede enviar tanto a un grupo de clientes[19] como a todos los que tenga registrados.

Para terminar con este apartado veamos dos operaciones más que puede realizar GRR, los crons y los python_hacks.

[17] Aquellas con el parámetro Request_id = 1.
[18] Esta etiqueta es alfanumérica y elegida por el usuario encargado del servidor, por lo que no es aleatoria.
[19] Estos clientes son identificados con la misma etiqueta.

3.2.3 Cron

Definición (Cron): Un cron no es más que la monitorización del envío periódico de hunts hacia equipos clientes. Se puede establecer cada cuanto tiempo se desea el lanzamiento de dichas peticiones y en qué instante se quiere dejar de enviarlas[20].

3.2.4 Python_Hacks

Definición (Python_Hack): Un python_hack es una herramienta que permite ejecutar código[21] en cualquier equipo final registrado en GRR. Para su funcionamiento primero subimos al servidor el script realizado en python. Tras ello lanzamos el flow o hunt de tipo ExecutePythonHack[22] indicándole el fichero '.py' y listo. Si queremos utilizar más veces ese mismo script no hará falta volver a realizar el primer paso, ya que se conserva en el servidor.

3.3 Estructura de GRR

Uno de los requisitos fundamentales de esta plataforma de análisis forense es que sea escalable a miles de máquinas finales, además de ser analizadas concurrentemente. En este apartado vamos a describir los diferentes componentes de GRR que hacen que todo esto sea posible y cómo puede llegar a ser desplegado en un clúster de servidores.

3.3.1 Cliente

El agente GRR es instalado en un equipo final. Desde ahí esperará a que le lleguen peticiones por parte del servidor, momento en el que las ejecutará y recabará la información resultante en un fichero CSV[23]. Finalmente, este documento será enviado al servidor GRR.

Esta comunicación cliente – servidor se realiza a través de peticiones HTTP POST, las cuales se distribuyen (a través del Load balancer[24]) a los servidores HTTP.

3.3.2 Servidores HTTP (Fronted Server)

Éstos se encargan de descifrar las respuestas POST de los clientes, empaquetarlos y encolarlos en una base de datos RDF. Básicamente permite que le llegue al servidor GRR todas las respuestas[25] de las peticiones lanzadas. Una vez que éstas son recibidas se contactará al worker[26] para procesar esa información.

Al igual que hace esta tarea, también recoge de las colas las peticiones del servidor para enviárselas a los clientes.

[20] Los crons no son obligatorios de usar. Se puede realizar el envío de flows y hunts esporádicamente sin establecer periodicidad.
[21] Código realizado en el lenguaje de programación Python.
[22] Petición que ejecuta código Python en el equipo cliente que le indiquemos.
[23] Los ficheros CSV son aquellos que guardan los datos en forma de tabla, diferenciando las columnas por comas. Se recomienda la apertura de este tipo de documentos mediante una hoja de cálculo para mejorar su visualización.
[24] Balanceador de carga.
[25] Procedentes de los equipos finales.
[26] Explicaremos su significado en la siguiente página.

3.3.3 Base de Datos RDF

Esta base de datos actúa tanto como un elemento de almacenaje central de datos[27] como un mecanismo de comunicación de todos los componentes.

3.3.4 Consola

GRR dispone tanto de una interfaz gráfica de usuario (GUI Console) como de una interfaz basada en texto o línea de comandos (CLI Console). Ambas sirven para que el analista forense interactúe con el sistema, enviando las peticiones contra los equipos clientes que desee y revisando los resultados obtenidos para llegar a dar un pronóstico.

3.3.5 Workers

Definición (Worker): Los workers se encargan de comprobar periódicamente si se han completado las peticiones. Cuando se recibe una petición completa este elemento recupera el flujo serializado de la base de datos.

En la siguiente imagen podemos ver de manera más clara las relaciones entre los distintos componentes que acabamos de explicar.

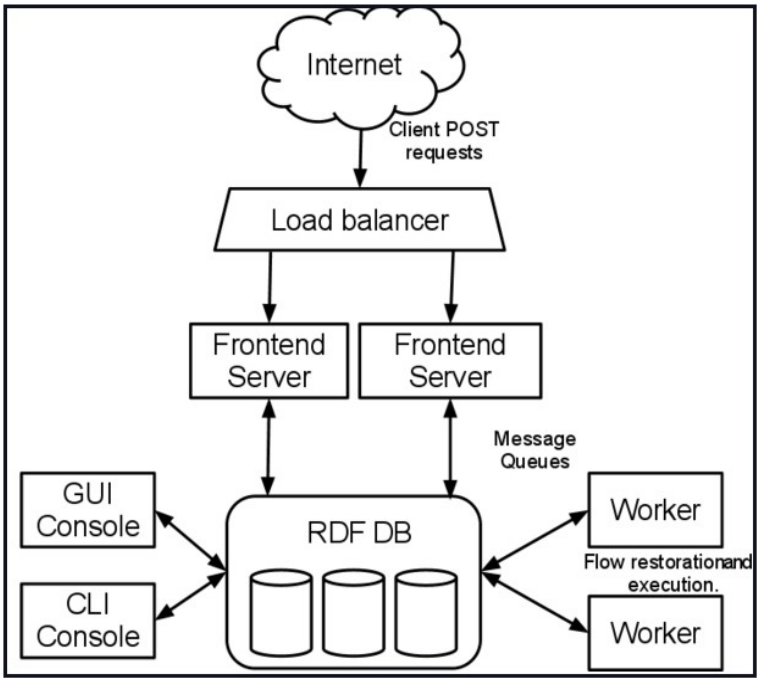

Figura 3.2 Arquitectura de GRR [10]

[27] Para el almacenamiento de los datos en esta BBDD se utiliza el modelo de datos AFF4.

3.4 Comunicaciones

Como hemos comentado anteriormente, el servidor GRR se comunica con los agentes instalados en equipos finales mediante parejas de mensajes “Petición – Respuesta” a través del protocolo HTTP POST.

3.4.1 Comunicación cliente – servidor

A continuación, explicaremos la comunicación resultante entre los diferentes elementos de GRR.

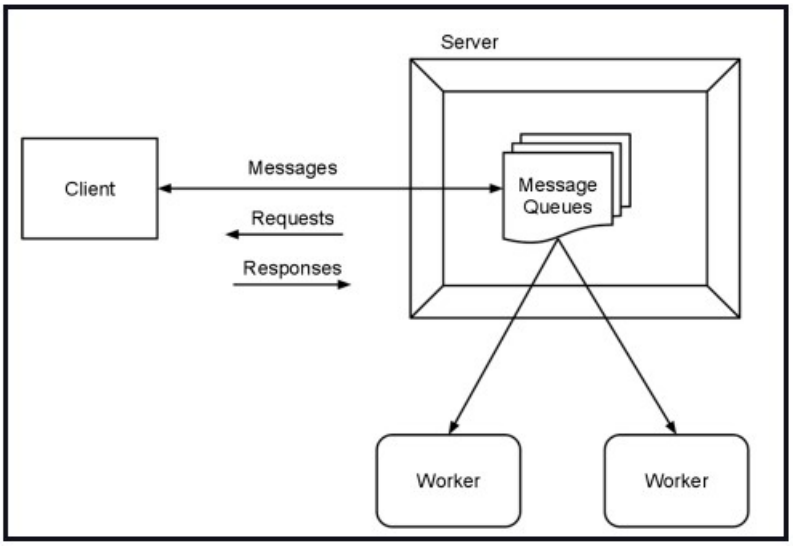

Figura 7.3 Comunicación entre los elementos de GRR [11]

El cliente recibe solicitudes por lotes y envía las respuestas por lotes igualmente. Desde el servidor estas peticiones son mandadas periódicamente[28] o cuando el analista lo requiera necesario.

Cada cliente tiene su propia cola de mensajes, desde las que recogen las peticiones que el servidor les envía. Además, éstos solo pueden recibir peticiones dirigidas hacia ellos, pero pueden enviar las respuestas a cualquier cola de “workers”. Si no tuviera cada cliente una cola no se podrían realizar consultas en paralelo, habría que esperar que acabara la solicitud para poder empezar otra.

3.4.1.1 Dirección de la comunicación

Hay que decir que siempre la comunicación va dirigida desde el servidor hacia el/los cliente/s, excepto cuando el cliente se registra por primera vez contra el servidor.

3.4.1.2 Client CN

El parámetro Client CN es un identificador que GRR da a los equipos registrados en él para su correcta identificación. Más adelante comentaremos como se forma este ID.

[28] Con la ayuda de los crons

3.4.2 Colas

GRR soporta múltiples colas de procesamiento en el servidor. En la siguiente imagen podemos ver que existen tres tipos:

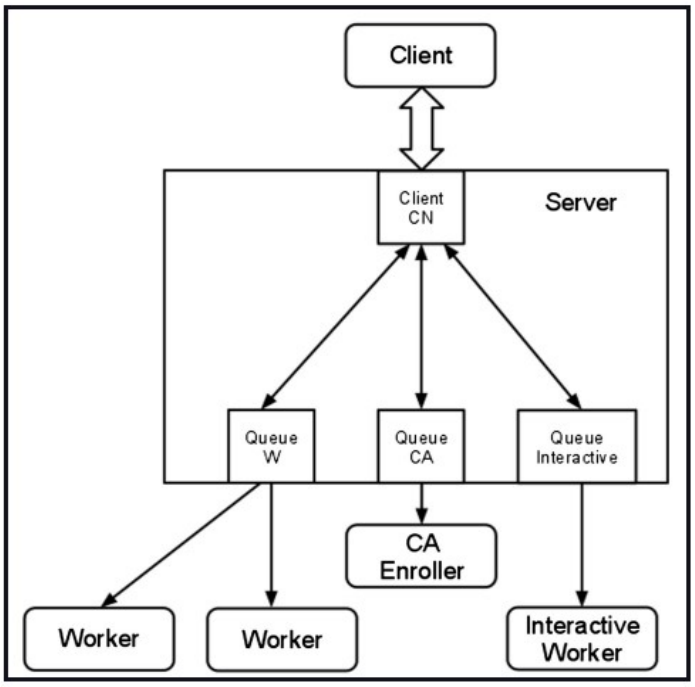

Figura 3.4 Tipos de colas en la arquitectura de GRR [12]

3.4.2.1 Cola W

Los workers que se encargan de esta cola recuperarán los mensajes que en ella se encuentren. Éstos corresponden a las respuestas de las solicitudes que el servidor previamente ha enviado al cliente.

3.4.2.2 Cola CA

En esta cola se guarda la información relacionada con la inscripción del cliente contra el servidor.

3.4.2.3 Cola Interactiva

Es la cola propia del cliente.

3.4.3 Registro de un cliente en el servidor

Todas las comunicaciones entre el servidor y el cliente son encriptadas usando AES256 con una clave de sesión aleatoria y un vector de inicialización que incluye el CSR[29].

Los clientes se inscriben automáticamente creando un CSR. Para ello el solicitante genera primero un par de claves (pública – privada).

[29] CSR ó Certificate Signing Request es un fichero de texto creado en el cliente para ser usado por el servidor con el fin de generar un certificado SSL, brindando de esta manera seguridad entre su comunicación.

En el primer intercambio de mensajes la comunicación la empieza el cliente, que es el que se va a registrar contra el servidor por primera vez. En dicho mensaje se encripta el CSR con la clave pública del cliente (para evitar que sea suplantado). Una vez en el destino, el certificado se guarda en la base de datos y será utilizado por el servidor para futuras conexiones con el cliente.

Ahora el siguiente paso lo realiza el servidor, que emite el certificado SSL generado y, además, envía un flow de tipo Interrogate[30].

Por último, el cliente responde a la petición.

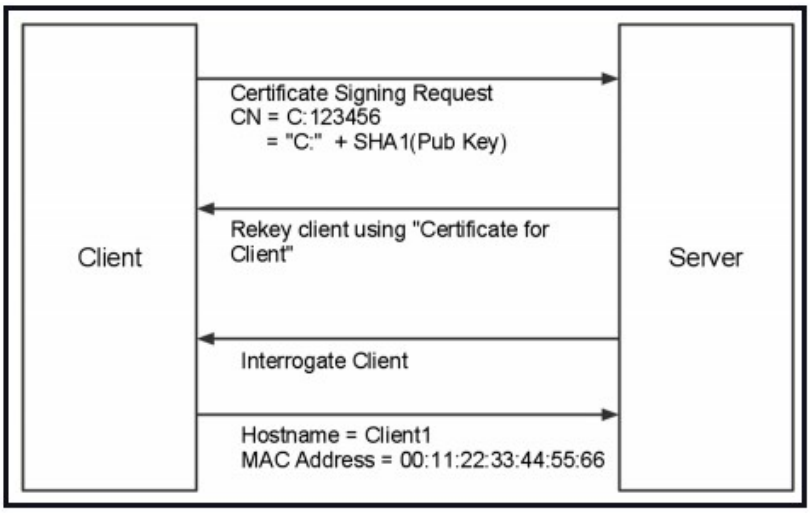

Figura 3.5 Inscripción del cliente contra el servidor GRR [13]

[30] Petición que sirve para obtener toda la información del cliente, como su IP, MAC, SO, preprocesador, etc...

4. SISTEMA CENTRALIZADO PARA EL CONTROL DE INTEGRIDAD: RBCHANGES

"No puedes preguntar a la gente qué es lo que quiere y después intentar dárselo. Para cuando lo hayas desarrollado querrán algo nuevo y diferente"

- Steve Jobs –

En el tema de introducción hemos hablado tanto de GRR como de un sistema capaz de detectar cambios de estado en equipos finales. Desde entonces os habréis estado preguntando a que nos referíamos con ambas cosas. Pues bien, una vez despejadas las dudas en el capítulo anterior sobre qué es GRR[31] y cómo funciona, ya podemos adentrarnos en profundidad en la herramienta que hemos logrado crear con la ayuda de este software: rbChanges.

4.1 Uso de GRR en el proyecto

De por sí con GRR tenemos un sistema de petición-respuesta manual para el análisis de uno o varios equipos. Este informe ayuda a la detección de malware en el sistema final, incluso podríamos diagnosticar de qué tipo es en concreto conociendo su comportamiento, como los procesos que ha abierto, las IP's con las que se ha comunicado, etc. Si pensamos detenidamente tenemos frente a nosotros una gran herramienta de explotación con la que "jugar" y poder crear una potente solución frente al malware y su propagación en un ámbito conocido compuesto por varios equipos.

Esto es precisamente lo que ha motivado a este proyecto: la creación de una herramienta que, basándose en GRR, consiga un sistema de detección de elementos sospechosos en uno o varios equipos finales. En caso de hallazgo, hay que ofrecer la posibilidad de proteger los equipos infectados cortando toda comunicación[32] con ellos mientras el equipo se encuentre en cuarentena. Esto último lo veremos en detalle en el siguiente capítulo.

[31] Se recomienda la lectura de los apartados 3.1 y 3.2 del capítulo 3 antes de seguir en este capítulo.
[32] Tanto entrante como saliente

4.2 Prevención de equipos frente al Malware

Podríamos preguntarnos ¿es posible prevenir el Malware antes de que infecte nuestros equipos? Sí, es posible. Si analizamos los cambios de estado de los equipos cliente y observamos anomalías en su comportamiento, seremos capaces de actuar antes de que el Malware se haya propagado en nuestra red.

Este es el objetivo del software que hemos denominado rbChanges. Con él podemos detectar un cambio en el estado de la red o de los procesos abiertos en un equipo final[33]. Gracias a este sistema de detección podríamos descubrir la presencia de un elemento sospechoso en el EndPoint.

Hay que aclarar que no siempre que haya un cambio de estado significa hallazgo de malware, pero si el usuario lleva un ritmo monótono en el sistema y es alertado por rbChanges podrá consultar los informes y ver si hay algo inusual, o bien, alguna actividad desconocida no haya realizado por sí mismo.

4.2.1 Posibilidad de rbChanges + Antivirus

Sin ningún indicio visual es difícil que un usuario se percate que un malware haya invadido su equipo, pero gracias a rbChanges puede ser alertado antes de que llegue a darse cuenta. Si además acompañamos esta herramienta con un software antivirus las posibilidades de detección y bloqueo de la propagación del malware se multiplican, ya que éste[34] solo alerta del elemento sospechoso y lo mete en cuarentena, no indica el rastro que ha dejado, por lo que no sabremos qué barreras de seguridad habrá que aportar al equipo para protegerlo frente a una posible próxima presencia del mismo malware.

4.3 Funcionamiento de rbChanges

4.3.1 La función de GRR

Lo primero que hacemos es invocar al agente servidor de GRR para que lance dos hunts contra los EndPoints registrados en él: Netstat y ListProcesses. El primero de ellos informa acerca del estado de la red mientras que el segundo lo hace sobre los procesos que están corriendo en los equipos finales. Estos hunts van a llegar a todos los EndPoints registrados en el servidor GRR.

Hagamos un inciso para recordar que un flow era el envío de una petición a un solo equipo. Si tenemos, por ejemplo, 10 equipos para enviar una petición cualquiera tendríamos que ejecutar 10 veces la orden de envío del flow. Este proyecto está pensado para soportar alrededor de 20.000 equipos, por lo que no sería óptimo lanzar 20.000 flows para Netstat y 20.000 para ListProcesses. Es por ello que hacemos uso de los hunts[35], ya que solo hacen falta enviar 2 peticiones[36] en lugar de las 40.000 si utilizáramos los flows.

En la siguiente imagen podemos ver como el servidor GRR lanza los dos hunts. Esto hace que automáticamente se envíe a cada equipo un flow de Netstat y otro de ListProcesses.

[33] A partir de ahora también denominaremos equipo final como EndPoint.
[34] Nos referimos al antivirus.
[35] Recordemos que un hunt es el envío de un flow a todos los equipos a la vez.
[36] Una para Netstat y otra para ListProcesses.

Para simplificarlo hemos hecho el ejemplo con dos EndPoints.

Es desde rbChanges donde se manda la orden de envío de los hunts a GRR, pero al ser este último el que se encarga de hacerlo no lo hemos puesto en la siguiente imagen.

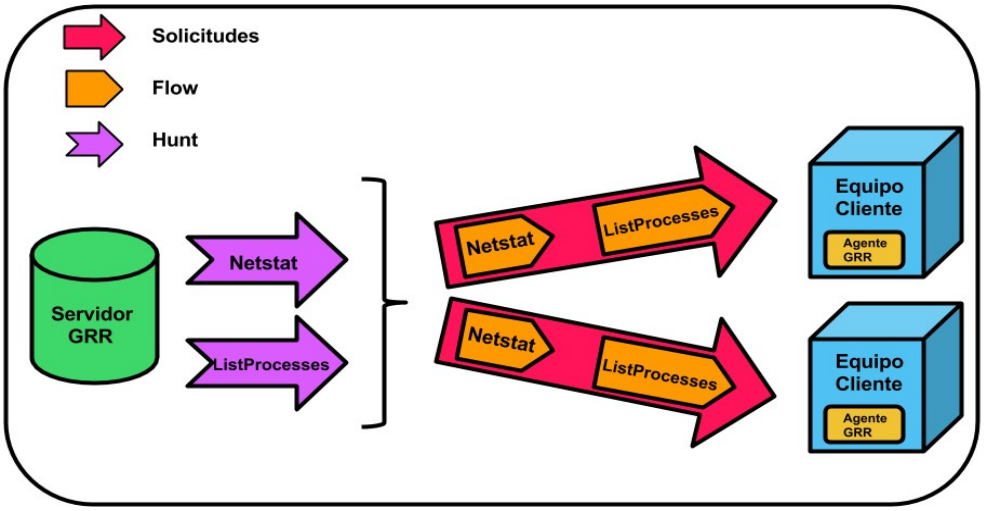

Figura 4.1 Ejemplo gráfico del envío de hunts a los EndPoints por GRR.

Una vez procesadas las solicitudes, los agentes de GRR enviarán los resultados al servidor a través de ficheros CSV. En cada EndPoint se generarán dos: uno referente a Nestat y otro a ListProcesses. Una vez que estos ficheros lleguen al servidor GRR se crearán dos CSV "gigantes"[37]. Uno englobará todos los resultados de Netstat de todos los equipos y el otro de ListProcesses.

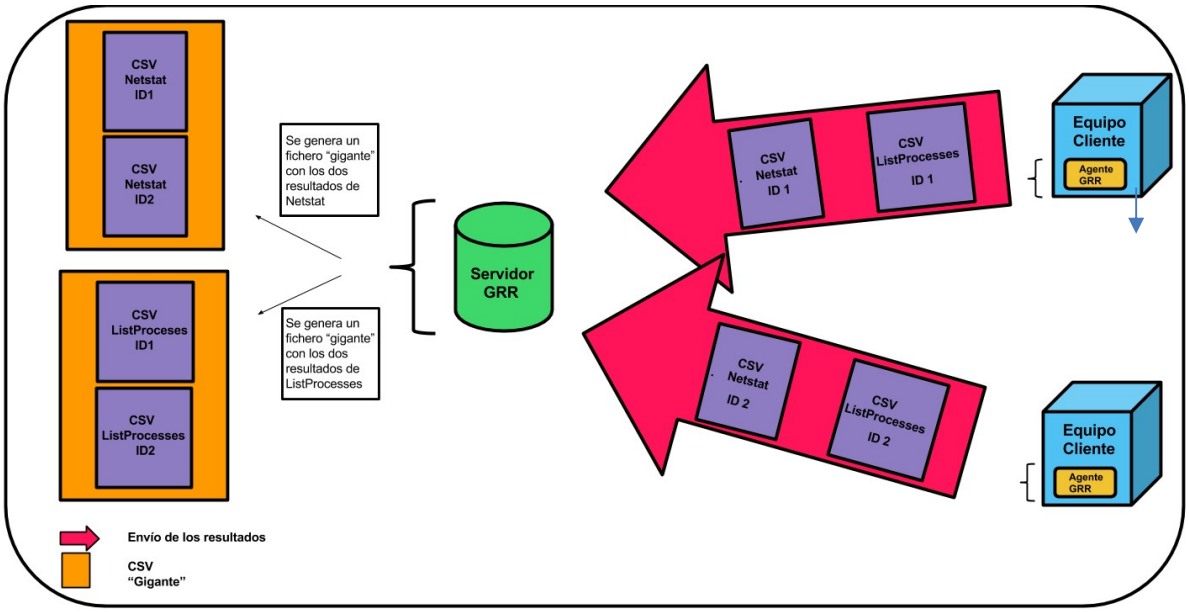

Figura 4.2 Ejemplo gráfico del envío de los resultados desde los EndPoints

[37] A partir de ahora denominaremos a los "CSV Gigantes" como CSV^{G}. Para referirnos a los de Netstat será CSV^{GN} y a los de ListProcesses CSV^{GP}.

4.3.2 Parseo de los ficheros CSV[G]

Como consecuencia de ello, para poder analizar el estado de la red de cada cliente por separado habrá que parsear el CSV[GN] y separarlo en diferentes ficheros. Os preguntareis, ¿para qué juntarlos todos en un mismo fichero si luego van a separarse de nuevo? Pues la respuesta es que GRR hace esto siempre así, junta en un mismo fichero todos los resultados del hunt lanzado. Una vez que lo ha formado nos lo muestra. Como no es objeto de este TFG modificar GRR sino aprovechar su funcionamiento tendremos nosotros que hacer esta separación en diferentes ficheros para poder analizar el comportamiento de los EndPoints. Haremos exactamente lo mismo para proceder al estudio de los procesos que están corriendo.

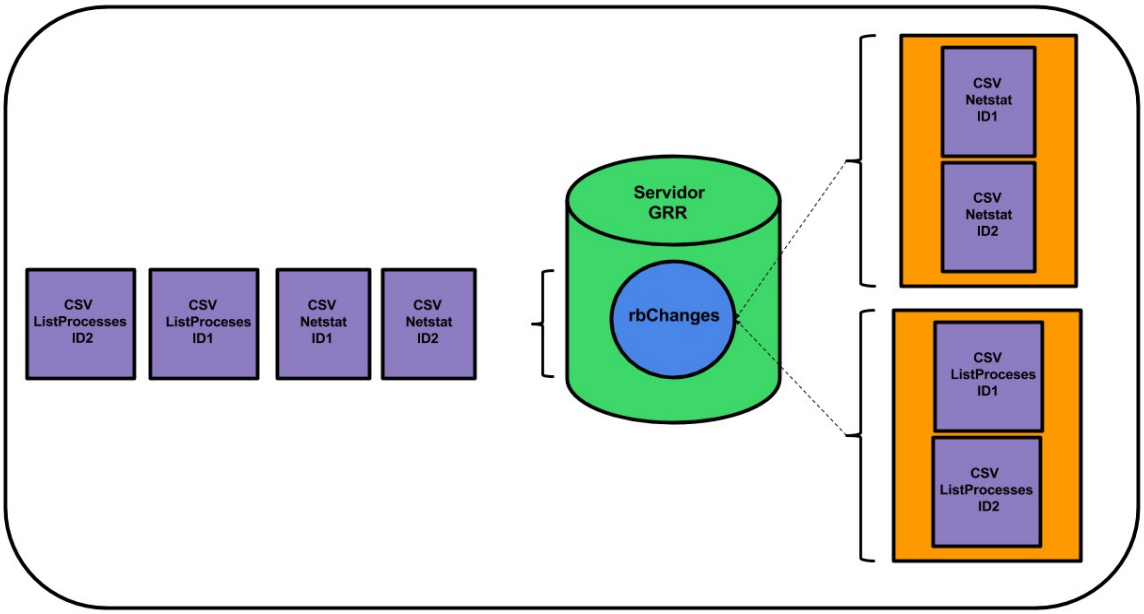

Figura 4.3 Creación de CSV individuales por rbChanges

Tras haber separado por EndPoints los dos CSV[G] podremos comenzar a analizarlos. El identificador será el que indique a qué equipo final pertenecen los ficheros CSV para que podamos diferenciarlos.

4.3.3 Método para determinar un cambio de estado

Antes de seguir aclaremos esto: Para determinar un cambio de estado en un EndPoint lo que haremos es, simplemente, comparar CSV's. Cada cierto tiempo[38] lanzaremos los dos hunts y recogeremos los resultados. Los CSV que se compararán serán los correspondientes a un tiempo X y X+Y.

[38] Periódico.

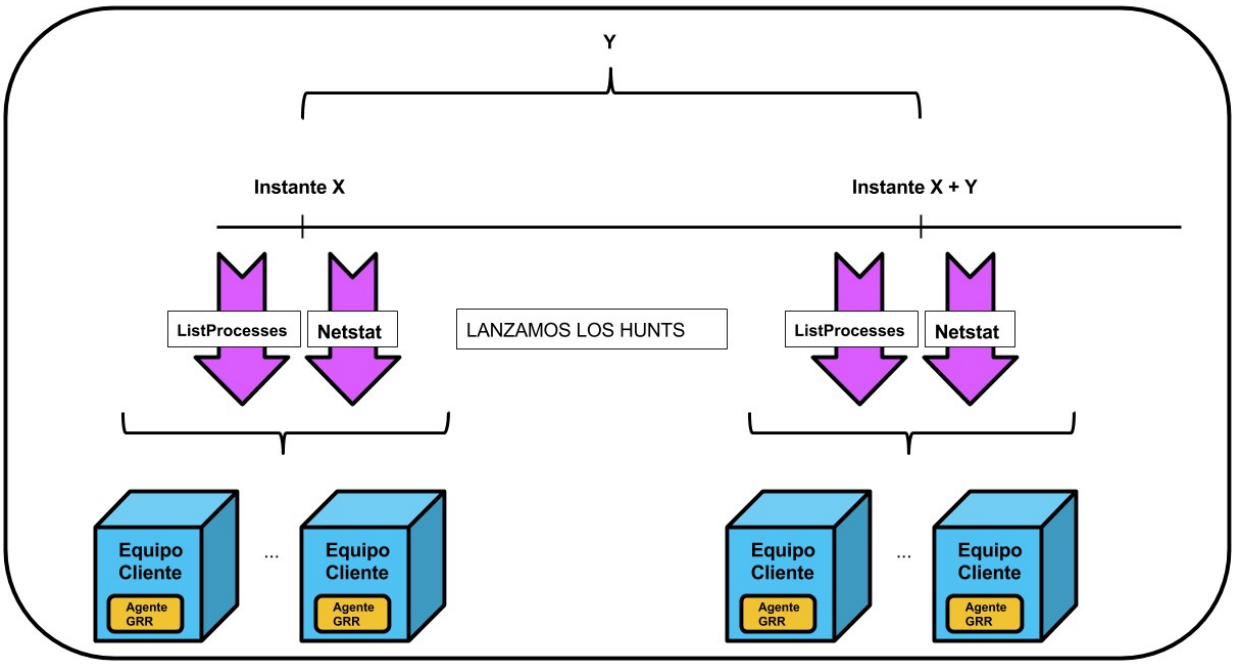

Figura 4.4 Envío periódico de los hunts. Este gráfico está simplificado.

En los CSV hay columnas que cambian siempre sus valores en cada ejecución del hunt, dando de esta forma falsos positivos en un cambio de estado. A la hora de analizarlos no nos interesa esto, por lo que ahora parsearemos también esos ficheros "individuales"[39] que acabamos de crear, extrayendo de ellos sólo las columnas consideradas como "estáticas". Un ejemplo de esto son aquellas que indican la hora en la que se ha realizado el análisis (entre otras). Siempre van a cambiar de un hunt a otro, por lo que tenemos que eliminarlas para comparar solo aquellas que nos den información relevante.

Así pues, crearemos otro CSV en el que sólo van a aparecer las columnas que determinen si, al variar de un instante de tiempo a otro, hay un cambio de estado. A este lo denominaremos CSV Resumido[40]. Hay que examinar cada CSV de cada tipo de hunt y estudiar cuáles son las columnas que nos interesan conservar para establecer un cambio de estado, ya que el CSV de Netstat no va a ser igual que el de ListProcesses (cada uno tiene unos parámetros distintos y, por consiguiente, unas columnas diferentes también).

En la siguiente imagen veremos este paso realizado para entenderlo mejor.

[39] Para referirnos a los ficheros "CSV Individuales" serán CSV^{I}, CSV^{IN} y CSV^{IP}.
[40] Para referirnos a los ficheros "CSV Resumidos" serán CSV^{R}, CSV^{RN} y CSV^{RP}.

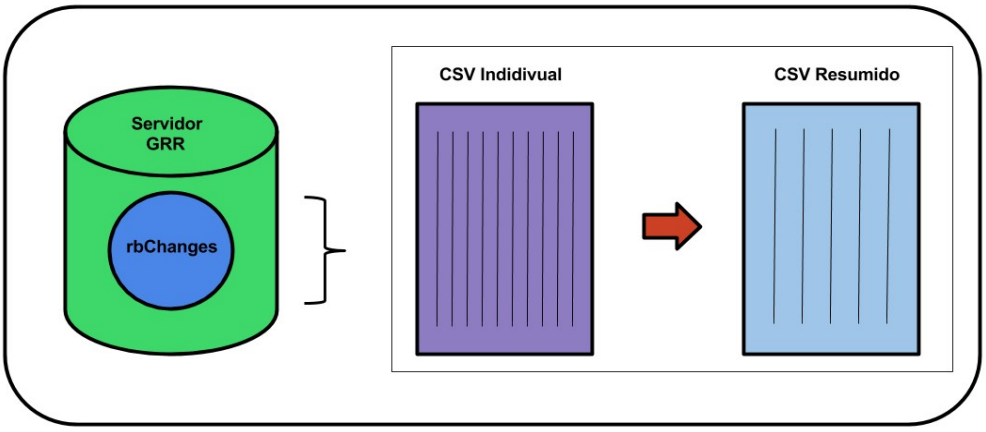

Figura 4.5 Proceso de resumen de los CSV' por rbChanges

Una vez que tenemos los CSV^R el siguiente paso es comprobar si hay o no un cambio de estado. Para ello se ha pensado en hacer uso de la función hash md5, debido a que identifica con los mismos caracteres alfanuméricos dos o más ficheros iguales. De esta forma podremos comparar los CSV^R obtenidos en los instantes X y X+Y. Si tienen el mismo hash significa que no ha habido cambio de estado, mientras que si no coinciden indica lo contrario, ya que hay algún parámetro que ha variado de un instante a otro.

4.3.4 La función de Aerospike

Una vez que hemos realizado el hash a un CSV^R entra en juego la base de datos Aerospike[41], en la que vamos a tener dos entradas como máximo por cada EndPoint (una que referencie a Netstat y otra a ListProcesses). Al principio la BBDD está vacía. Se comenzará a rellenar cuando lleguen los resultados de los hunts y les hagamos el hash.

Para seguir explicando pongamos de ejemplo que hemos instalado rbChanges y lo arrancamos. Se lanzarán por primera vez el hunt de Netstat y el de ListProcesses en el instante X. Supongamos que han llegado primero los resultados de este último. Se establecerá una entrada en la tabla[42] para el equipo con ID1 y el hunt de procesos; y otra para aquel con ID2 (con hashes 5678 y 2895 respectivamente, p.e). Ahora llega el CSV de Netstat. Tras realizarle el hash se introduce en la tabla una entrada para el equipo con ID1 y hunt Netstat y otra para aquel con ID2 (con hashes 1234 y 4632 respectivamente, p.e).

[41] La explicamos detalladamente en el apartado de Arquitectura.
[42] El nombre de la tabla es "grr", dentro del espacio de trabajo "malware".

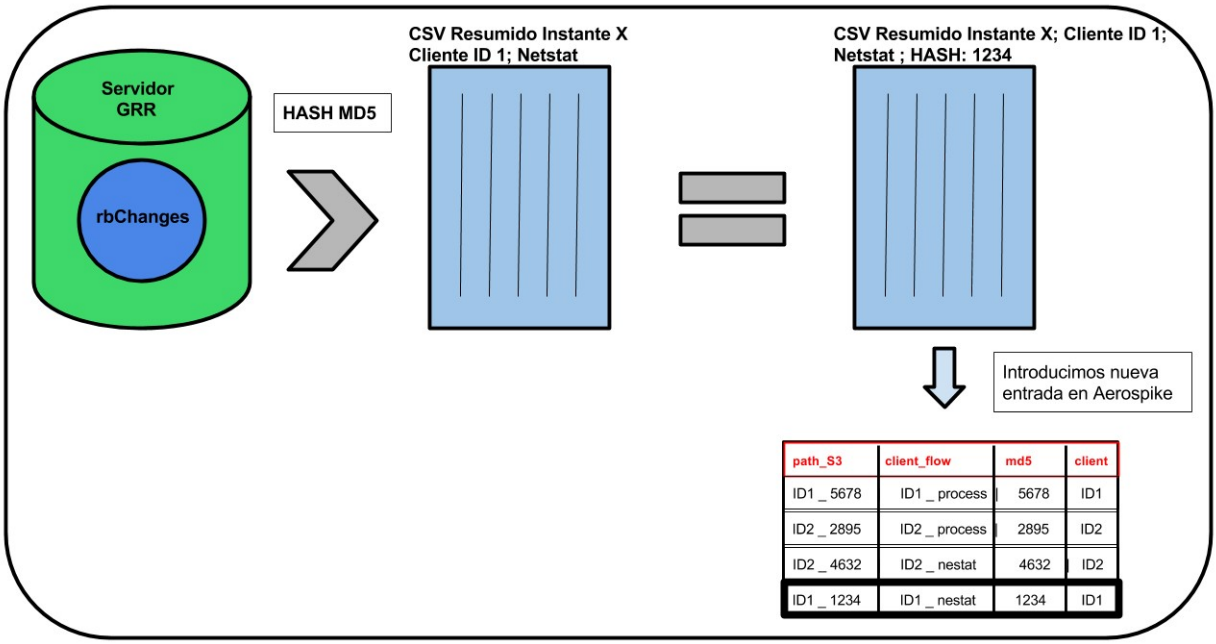

Figura 4.6 Ejemplo de introducción nueva entrada en Aerospike

Pasado un tiempo 'Y', rbChanges vuelve a lanzar los dos hunts. Como ya no es la primera vez que lo hacemos la tabla no va a estar vacía, sino que va a tener una entrada para cada EndPoint y cada hunt. Cuando lleguen ahora los nuevos resultados se buscará en la tabla la entrada correspondiente y se compararán los hashes para determinar si son o no iguales. Por ejemplo, se crea el CSV^{IN} del cliente con ID1. Ahora se modifica a CSV^{RN} y se le calcula el hash. Se busca en la tabla por la columna client_flow, que en este caso sería ID1_netstat. Cuando la encuentre mira en su misma fila el valor de md5. Si es igual al del CSV^{RN} que ha llegado en el instante X+Y es porque no ha habido cambio de estado, mientras que si es distinto sí. En este último caso se modificará la entrada antigua por la nueva.

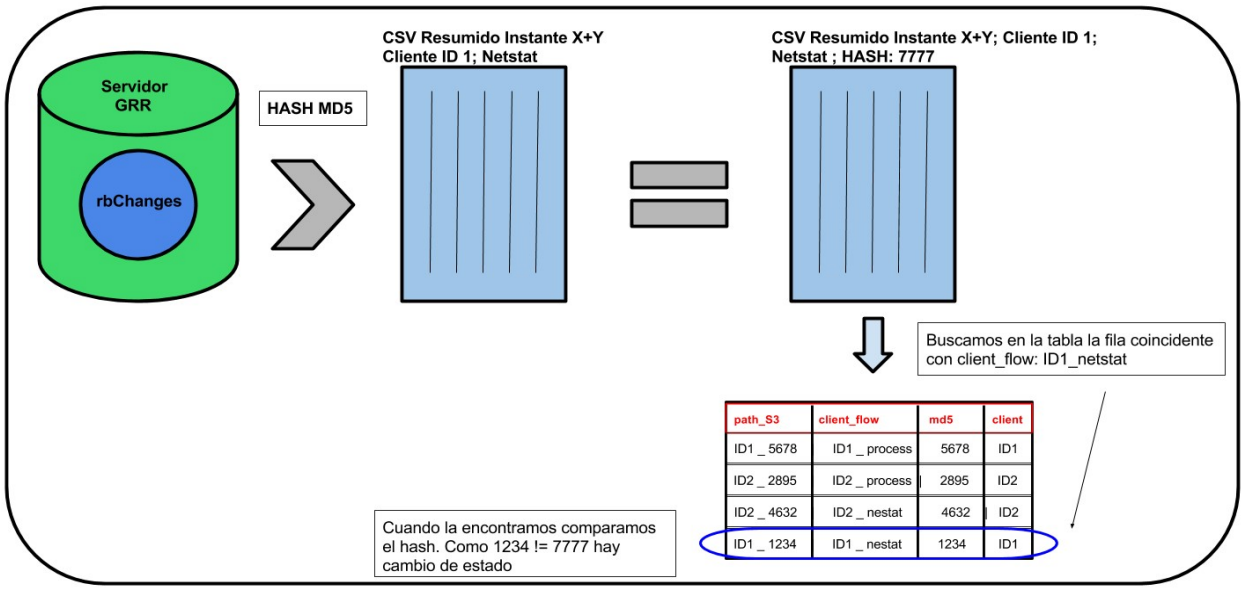

Figura 4.7 Ejemplo de CSV con diferentes hashes

Finalmente la tabla quedaría así:

path_S3	client_flow	md5	client
ID1 _ 5678	ID1 _ process	5678	ID1
ID2 _ 2895	ID2 _ process	2895	ID2
ID2 _ 4632	ID2 _ nestat	4632	ID2
ID1 _ 7777	ID1 _ nestat	7777	ID1

Figura 4.8 Ejemplo de modificación de la tabla por haber un cambio de estado

En caso de que coincidiera el hash no se modificaría.

4.3.5 La función de S3

El objetivo de esto es poder, desde la web del manager de redBorder, descargarse los dos ficheros CSV del cambio de estado (el anterior y el nuevo) para poder analizarlos. Aunque para comprobar si hay o no un cambio de estado hacemos uso de los ficheros CSV^R, los que se deben descargar desde la interfaz web son los antecesores a éstos (CSV^I). Para ello hay que subir a S3 estos ficheros cada vez que se cree o modifique una entrada en Aerospike. La columna path_S3 indica el nombre dado a los CSV^I subidos a S3.

4.3.6 La función de kafka

Cuando se detecta un cambio de estado, además de hacer esto que hemos comentado en Aerospike, también se lanzará un mensaje a Kafka indicando que ha habido un cambio de estado, con el topic rb_changes.

4.4 Arquitectura

rbChanges necesita de una serie de componentes adicionales para asegurar la efectividad de su funcionamiento. A continuación, mostraremos su escenario y explicaremos cada uno de los componentes:

4.4.1 Endpoints

Son los equipos finales que van a tener instalado el agente cliente de GRR.

4.4.2 rbChanges

Software incluido en el servidor GRR encargado de controlar si ha habido algún cambio de estado en los sistemas finales.

4.4.3 Agente GRR

Es la parte de GRR que se instala en el cliente para realizar toda comunicación con el servidor. Su instalación se podrá hacer de forma manual, descargándose desde la interfaz web del Manager de redBorder.

4.4.4 Kafka

Cola de mensajes a la que van a llegar eventos por parte de rbChanges cuando se produzca un cambio de estado en algún Endpoint para avisarnos. En dicho mensaje, con formato json, se informará cuáles son el viejo y el nuevo CSV[I] que contienen el estado de red o de procesos del EndPoint (dependiendo cuál de ellos sea el que haya cambiado).

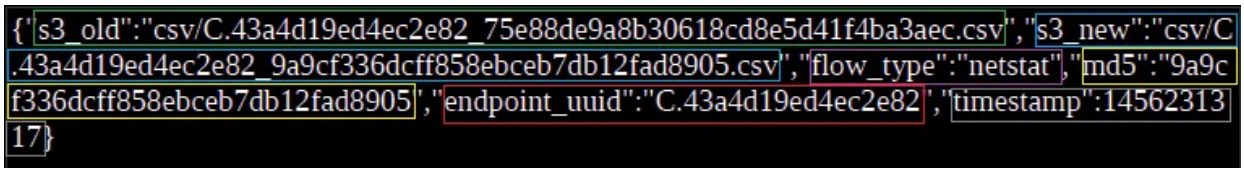

Figura 4.9 Ejemplo de mensaje que recibe kafka

4.4.5 S3

Almacenamiento de objetos donde van a ser guardados los ficheros CSV[I] de tipo Nestat o ListProcesses. Éstos se podrán descargar desde la interfaz web del Manager de RedBorder. Desde aquí se obtendrán tantos los antiguos como los nuevos CSV[I] de un determinado hunt.

4.4.6 Aerospike

Esta base de datos va a ser utilizada por rbChanges para almacenar en ella una serie de parámetros útiles para determinar un cambio de estado en los EndPoints:

4.4.6.1 s3_path

Indica la ruta en S3 de los ficheros CSV[I]. La terminación de éstos es con extensión '.csv'. Recordemos que los ficheros aquí guardados no son los "resúmenes", sino los "enteros individuales". Los CSV[R] sólo se usaban para calcular el hash, pero los que luego nos descargamos desde la web del Manager para analizarlos son los CSV[I].

4.4.6.2 client_flow

Guarda la unión del identificador del cliente con el tipo de flow[43]. Esta es la clave primaria de la tabla, ya que sus valores son todos distintos siempre. Además, será en ellos donde buscaremos la entrada correspondente del nuevo CSV[R] que ha llegado y ver si el hash es el mismo o no[44].

En esta tabla un mismo cliente podrá tener como máximo dos entradas, ya que sólo se utilizarán dos tipos de flows.

4.4.6.3 md5

Contiene el hash del fichero. Este parámetro será el que miremos para saber si ha cambiado algo

[43] Netstat o ListProcesses.
[44] Recordemos que así determinábamos los cambios de estado. Si son iguales los hashes no hay cambio de estado y si son diferentes sí.

de un CSV[I] a otro, es decir, para afirmar que ha habido un cambio de estado en algún equipo.

4.4.6.4 client

Es el identificador que GRR da a sus clientes.

A continuación, mostramos un ejemplo de esta tabla:

```
aql> select * from malware.grr
+-----------------------------------------------------+-----------------------------+----------------------------------+---------------------+
| s3_path                                             | client_flow                 | md5                              | client              |
+-----------------------------------------------------+-----------------------------+----------------------------------+---------------------+
| "C.0c52ed527cfa5afd_b985e6005e0e715eeb7abe94aa10e081.csv" | "C.0c52ed527cfa5afd_netstat" | "b985e6005e0e715eeb7abe94aa10e081" | "C.0c52ed527cfa5afd" |
| "C.909f3ba709e587ad_72d8d52a616edff74f9f97a8d6b9de39.csv" | "C.909f3ba709e587ad_netstat" | "72d8d52a616edff74f9f97a8d6b9de39" | "C.909f3ba709e587ad" |
| "C.43a4d19ed4ec2e82_123e9e96344d1848e79a4f54b6d38198.csv" | "C.43a4d19ed4ec2e82_netstat" | "123e9e96344d1848e79a4f54b6d38198" | "C.43a4d19ed4ec2e82" |
| "C.5143a262fdd8d921_51418425f7f7d6006dd6ac6287e4a06f.csv" | "C.5143a262fdd8d921_process" | "51418425f7f7d6006dd6ac6287e4a06f" | "C.5143a262fdd8d921" |
| "C.5143a262fdd8d921_bab2798105c130d2dd4574b65b3fcb38.csv" | "C.5143a262fdd8d921_netstat" | "bab2798105c130d2dd4574b65b3fcb38" | "C.5143a262fdd8d921" |
| "C.0c52ed527cfa5afd_fd59c527de7006d8ccbcfc13b04b3c07.csv" | "C.0c52ed527cfa5afd_process" | "fd59c527de7006d8ccbcfc13b04b3c07" | "C.0c52ed527cfa5afd" |
| "C.43a4d19ed4ec2e82_1ac4c31d778faea3ce31ad40fc25c5a2.csv" | "C.43a4d19ed4ec2e82_process" | "1ac4c31d778faea3ce31ad40fc25c5a2" | "C.43a4d19ed4ec2e82" |
| "C.909f3ba709e587ad_8f5befd30643d7216c2294f0e53b6b65.csv" | "C.909f3ba709e587ad_process" | "8f5befd30643d7216c2294f0e53b6b65" | "C.909f3ba709e587ad" |
+-----------------------------------------------------+-----------------------------+----------------------------------+---------------------+
9 rows in set (0.198 secs)
```

Figura 8.10 Ejemplo de Aerospike en rbChanges

Recordemos que las entradas se van actualizando con el hash del último fichero CSV generado. Si el hash es distinto, se ha producido un cambio de estado y se actualizan los campos correspondientes al cliente

4.2.7. Manager de redBorder

Puede ser tanto uno como un clúster de nodos. Al ser distribuido, tanto Kafka como Aerospike podrían estar presentes en más de uno de ellos.

Por último, mostramos gráficamente la arquitectura que acabamos de explicar.

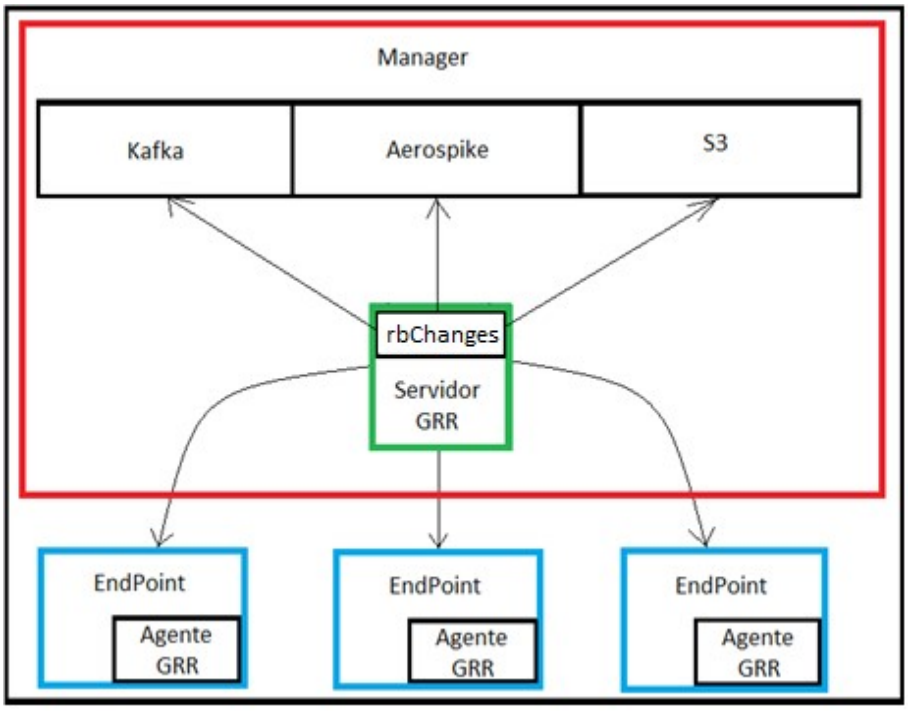

Figura 4.11 Arquitectura de rbChanges

4.3. Estructura

Para el correcto funcionamiento del software son necesarios una serie de ficheros con los que trabaja rbChanges para asegurar un funcionamiento óptimo. Veámoslos a continuación.

4.4.1 Scripts

- **class_grr.py**

  Fichero donde se encuentra la clase principal que da comienzo con el servicio rbChanges. Desde aquí podremos lanzar los hunts de Netstat y ListProcesses, recoger los CSV, introducir y consultar datos en Aerospike, comparar los hashes de los CSV para corroborar si existe o no un cambio de estado, mandar eventos a kafka informando acerca de ese cambio y subir a S3 los CSV.

  Su ruta completa es "~/grr_support.git/ grr_service/scripts/class_grr.py". En el Anexo A puede verse su contenido.

- **Directorio hunt_files**

  - **launch_netstat.py**

  Fichero en el que está incluido el código necesario para lanzar el hunt de tipo Nestat en los equipos clientes registrados en GRR. Este fichero será llamado por class_grr para ejecutarlo en la CLI Console de GRR.

  Su ruta completa es "~/grr_support.git/grr_service/scripts/hunt_files/launch_netstat.py". En el Anexo B puede verse su contenido.

  - **launch_process.py**

  Fichero con ruta ~/grr_support.git/grr_service/scripts/hunt_files/launch_process.py, en el que está incluido el código necesario para lanzar el hunt de tipo ListProcesses en los equipos clientes registrados en GRR. Este fichero será llamado por class_grr para ejecutarlo en la CLI Console de GRR.

  Su ruta completa es "~/grr_support.git/grr_service/scripts/hunt_files/launch_process.py". En el Anexo C puede verse su contenido.

- **Directorio config**

  - **parameters.json**

  Fichero JSON donde se guardan la clave de acceso y password para poder conectarnos con S3, el bucket donde se van a añadir los CSV en S3, el broker de kafka y los datos necesarios para conectarse a aerospike.

  Su ruta completa es "~/grr_support.git/ grr_service/scripts/config/parameters.json". En el Anexo D puede verse su contenido.

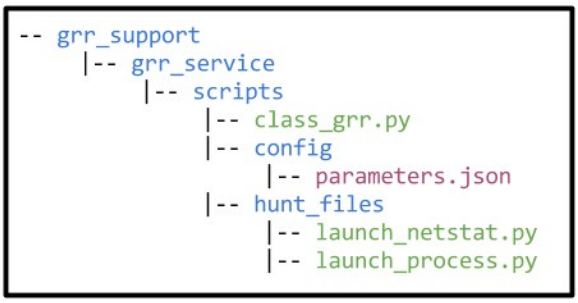

Figura 4.12 Estructura de ficheros 1

4.4.2 Fichero de logs

- **status_changes**

  Fichero .log en el que se van a registrar todos los logs generados por el servicio rbChanges. Se encontrará en la ruta "/var/log/grrstates/status_changes.log".

4.4.3 Temporal

- **Directorio /tmp/grrstates/temp_csv/hunts**

  Espacio de trabajo en el que se exportan los resultados de los hunts. En él se creará un directorio por cada hunt lanzado (denominado igual que el identificador[45] del hunt). Cada uno contendrá un CSV[G] compuesto por los resultados de todos los clientes que han ejecutado dicho hunt. Recordemos que ese CSV[G] será parseado más adelante para separar los resultados por cliente y, así, comprobar si ha habido un cambio de estado en alguno de ellos.

- **Directorio /tmp/grrstates/change_state/hunts**

  El fichero class_grr creará un directorio aquí por cada cliente existente registrado en GRR. Estos directorios se llamarán igual que el identificador que GRR le ha dado a cada uno de los EndPoints.

  - **Directorio cuyo nombre corresponde con el ID de un equipo final**

    En él se van a crear dos directorios más: uno para el flow Netstat y otro para el de ListProcesses. En la imagen que veremos a continuación este directorio se correspondería con el ID 'C.0c52ed527cfa5afd'.

    - **netstat**

      Directorio en el que se van a guardar los CSV[I] del flow Netstat de dicho cliente. Desde aquí se subirán a S3 los ficheros que determinen que ha habido un cambio de estado respecto el anterior.

    - **process**

      Directorio en el que se van a guardar los CSV[I] del flow Netstat de dicho cliente. Desde aquí se subirán a S3 los ficheros que determinen que ha habido un cambio de estado respecto el anterior.

[45] Recordemos que este parámetro se llamaba Request_id. Para más información consultar el apartado 3.2.1.1 del capítulo 3.

- **hunt_nestat.txt**

  Fichero de texto temporal en el que el fichero launch_netstat escribe el identificador del hunt que ha lanzado, necesario para la exportación del CSV por class_grr.

  Su ruta completa es ~/grr_support/grr_service/change_state/hunts/hunt_netstat.txt

- **hunt_process.txt**

  Fichero de texto temporal en el que el fichero launch_process escribe el identificador del hunt que ha lanzado, necesario para la exportación del CSV por class_grr.

  Su ruta completa es ~/grr_support/grr_service/change_state/hunts/ hunt_process.txt

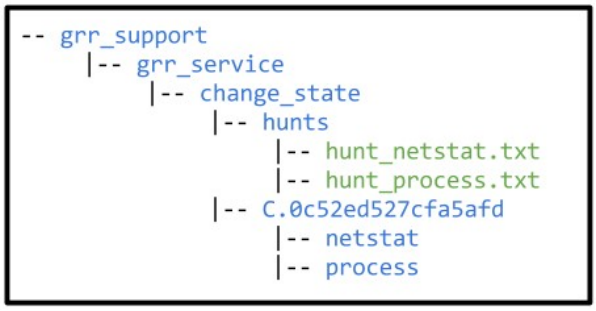

Figura 4.13 Estructura de ficheros 2

4.4. Conclusiones

Como hemos podido ver hasta ahora, no estamos hablando de un antivirus al uso ya que rbChanges no está al pendiente de las actividades en tiempo real que ocurren en el equipo, sino que el agente GRR instalado en los EndPoints responde a las peticiones que le envía el servidor cada cierto tiempo.

Esa es la principal característica que lo diferencia de un antivirus convencional. Por supuesto, no es un sustituto de éste, es un complemento adicional que nos ayuda a seguir el rastro dejado por cualquier malware que haya infectado un equipo final. Por ejemplo, analizando el estado de la red podríamos averiguar las conexiones con las que se habría estado comunicando el malware. De esta manera tomaríamos las medidas oportunas para proteger el equipo frente a esas direcciones IP. En caso de que de nuevo ese malware intentara atacar al sistema final ya estaríamos inmunizados.

5. SISTEMA DE MITIGACIÓN DE INTRUSIONES: RBCONTENTION

"Pienso que los virus informáticos muestran la naturaleza humana: la única forma de vida que hemos creado hasta el momento es puramente destructiva"

- Stephen Hawking -

Ya hemos estado hablando en el capítulo anterior acerca de rbChanges y su sistema de detección malware en equipos finales. Por fin sabemos cómo funciona esta herramienta y qué es lo que nos aporta. Ahora es el turno de complementarla con otras características para que, durante el análisis de los equipos comprometidos, los EndPoints no tengan ninguna posibilidad de comunicarse con nadie que no queramos. Con esta medida de protección nos aseguramos que, mientras se confirme cualquier presencia malware, estos equipos no estén expuestos a "los malos".

5.1 Contención de EndPoints: rbContention

Esta herramienta es la encargada de bloquear todas las conexiones entrantes y salientes de un EndPoint, excepto aquellas que se le indiquen explícitamente. La única excepción es que no va a perder comunicación con el Manager nunca. Si ésta se corrompiera no podríamos desbloquear esta funcionalidad. La única manera de volver a tener conexión con el exterior sería modificar manualmente el firewall de Windows, cosa que no todo el mundo sabe hacer.

Desde la web del Manager tenemos disponible esta opción. Es también desde ahí donde introduciremos las IP's con las que queramos seguir manteniendo contacto mientras perdura el análisis (y el tiempo que el analista crea conveniente en el caso de llegar a encontrar amenaza malware). No hará falta que la IP del Manager la indiquemos, ya que implícitamente lo hace rbContention.

5.2 Estructura de rbContention

Antes de explicar el funcionamiento de esta aplicación veamos los ficheros de los que se compone.

5.2.1 Scripts

- **block_endpoint.py**

  Fichero encargado de ejecutarse en el EndPoint para bloquearle toda conexión con el exterior (excepto con las direcciones que se le indiquen).

  Su ruta completa es "~/grr_support/python_hacks/firewall/block_endpoint.py". Puede verse su contenido en el Anexo E.

- **unblock_endpoint**

  Fichero que se ejecutará en el EndPoint para restaurar las conexiones del equipo a su estado anterior (antes de ejecutarse block_endpoint).

  Su ruta completa es "~/grr_support/python_hacks/firewall/unblock_endpoint.py". Puede verse su contenido en el Anexo F.

```
-- grr_support
    |-- python_hacks
        |-- firewall
            |-- block_endpoint.py
            |-- unblock_endpoint.py
```

Figura 5.1 Estructura de fichero de rbContention

5.3 Funcionamiento de rbContention

Esta herramienta hace uso de python_hacks[46] para ejecutar en los EndPoints los scripts anteriores. En ellos se encuentra la lógica para bloquear y desbloquear toda conexión con el exterior de los equipos finales.

5.3.1 Bloqueo de EndPoints

Para cortar toda conexión entrante y saliente de un EndPoint esta aplicación hace uso del firewall de Windows.

Primero enviará el script block_endpoint a través del flow ExecutePythonHack al EndPoint. A éste se le puede acompañar de una serie de parámetros en la petición que, en nuestro caso, serán las direcciones IP con las que queramos seguir manteniendo conexión una vez bloqueado el sistema final.

[46] Explicado en el apartado 3.2.4 del capítulo 3.

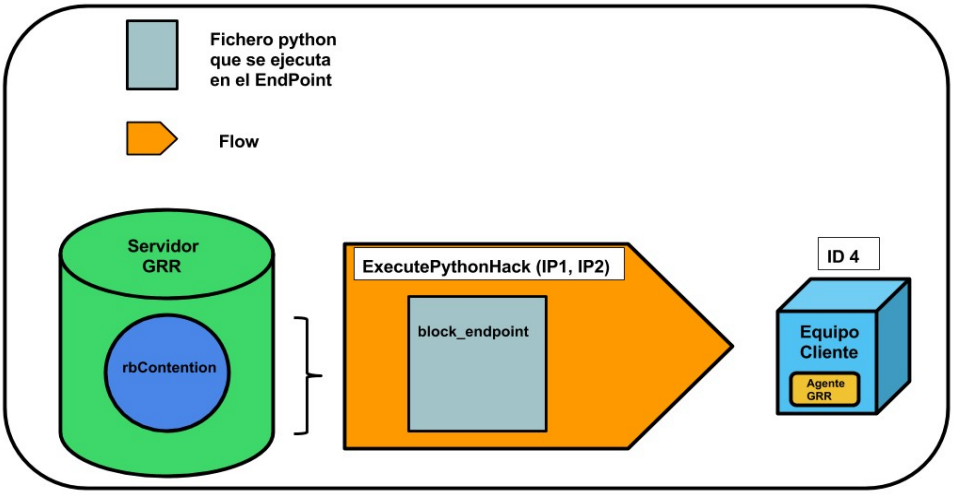

Figura 5.2 Envío del flow ExecutePythonHack para el bloqueo del EndPoint

Una vez que el fichero se encuentre en el EndPoint, se ejecutará automáticamente para modificar las reglas del firewall. Comenzará mandando la orden de bloqueo de todas las conexiones entrantes y salientes del equipo. Una vez hecho esto se permitirán las conexiones bidireccionales con las IP's indicadas.

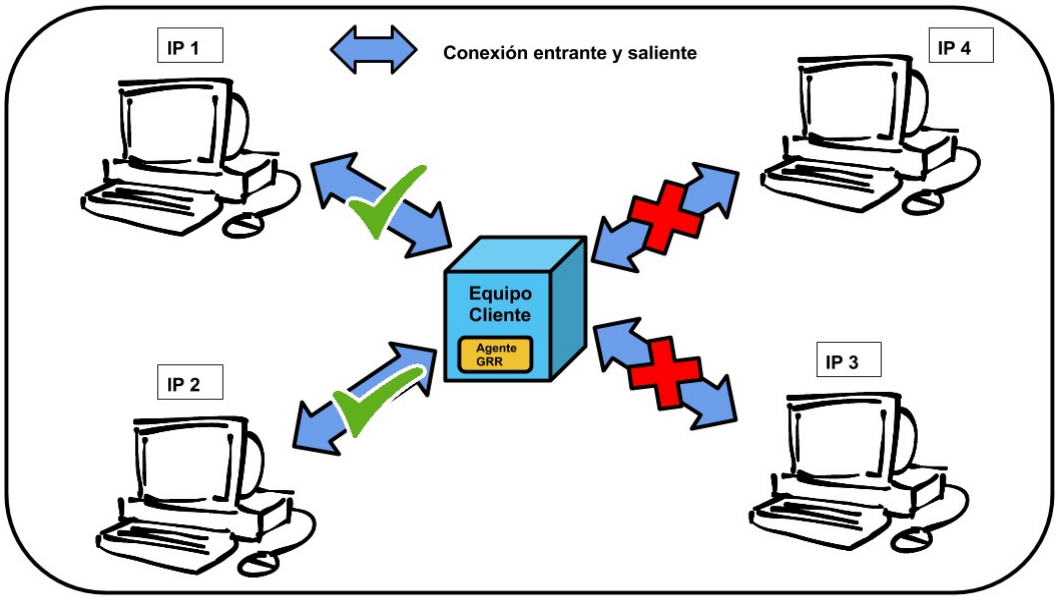

Figura 5.3 Bloqueo realizado

Comentar que, aunque no esté ilustrado en la imagen anterior, siempre va a tener comunicación con el servidor GRR. Si no tuviera sería un caos, ya que no se podría desbloquear. No es necesario que le pasemos la dirección IP de éste como parámetro ya que lo hace implícitamente.

5.3.2 Desbloqueo de EndPoints

Ahora enviaremos el script unblock_endpoint a través del flow ExecutePythonHack al EndPoint. No hace falta acompañarlo de ningún parámetro, ya que lo que hace este fichero es deshacer la acción anterior.

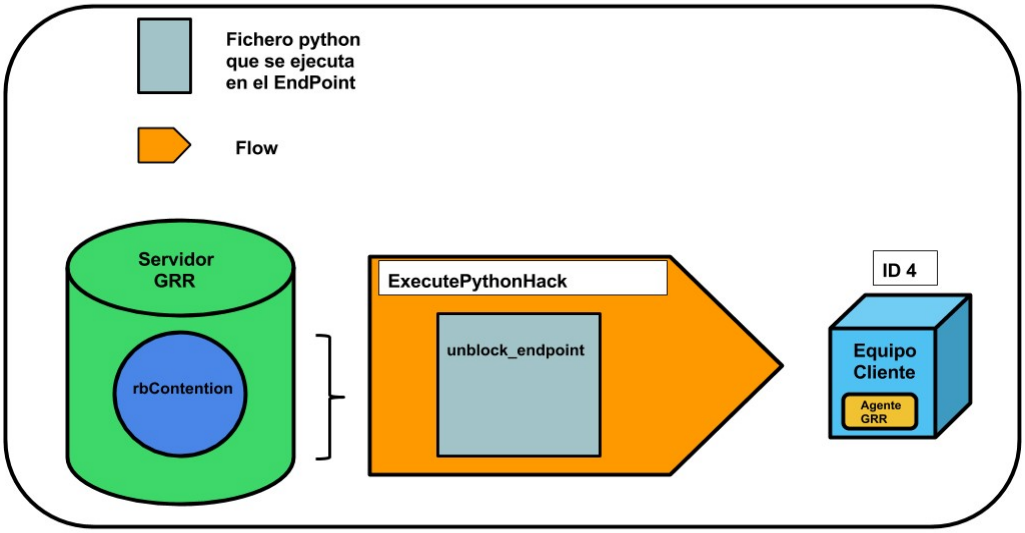

Figura 5.4 Envío del flow ExecutePythonHack para el desbloqueo del EndPoint

Una vez que el fichero se encuentre en el EndPoint, se ejecutará automáticamente para modificar las reglas del firewall. En este caso volverá a permitir conexión con el exterior.

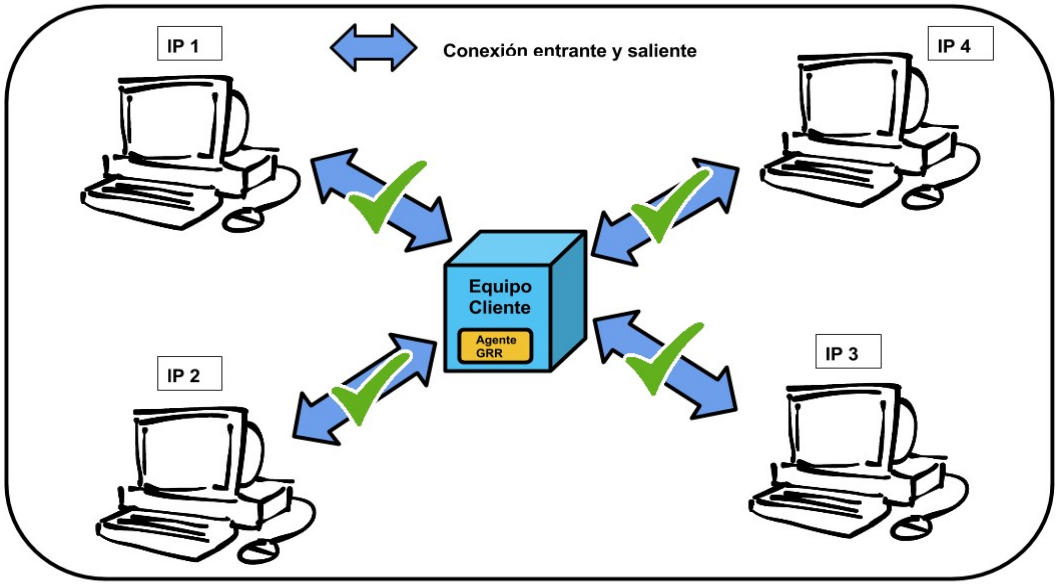

Figura 5.5 Desbloqueo realizado

En un capítulo posterior veremos como un usuario utiliza esta aplicación desde el proyecto redBorder Malware.

5.4 Conclusiones

Con esta aplicación conseguimos un mecanismo de seguridad en el caso de que se llegaran a encontrar indicios de intrusión en algún equipo cliente.

6. OTRAS FUNCIONALIDADES: INICIO/PARADA DEL SERVIDOR GRR Y AUDITORÍA

"Si amas lo que haces nunca será un trabajo"
- Confucio -

Para el mantenimiento de GRR, rbChanges y rbContention en el proyecto redBorder Malware ha sido necesaria la creación de dos herramientas adicionales. A continuación podremos descubrir de qué tratan.

6.1 Auditoría: rbAudit

Esta funcionalidad obtiene un registro de todo lo que se ha hecho con GRR (lanzamiento de hunts, crons, flows, nuevos registros de EndPoints, etc.). GRR guarda toda esa información en forma de logs, sólo que para poder obtenerla hay que hacer uso de la CLI Console, ya que se encuentra dentro del modelo de datos AFF4.

Con esta aplicación podremos obtener esos logs y plasmarlos en un fichero de texto para ser legible con mayor facilidad. De esta forma quitamos de por medio todos los comandos engorrosos que habría que ejecutar si se hiciera manualmente desde la consola de GRR.

6.1.1 Funcionalidad

Esta herramienta es útil para saber hallar o descartar errores sugidos en GRR.

Imaginemos, p.e, que queremos bloquear las conexiones de un equipo final y ejecutamos rbContention. Al cabo de un rato vemos que no se ha aplicado todavía en el EndPoint. Empezamos a sospechar que algo ha pasado, pero no sabemos si ha sido culpa de GRR, de nuestro programa o del sistema final. Una posibilidad que tenemos es mirar este registro de logs para comprobar si GRR ha enviado o no la petición correspondiente, descartando así opciones. En caso de que veamos que la solicitud se ha enviado correctamente deberíamos seguir investigando las otras posibilidades.

6.1.2 Estructura

Para su funcionamiento son necesarios una serie de ficheros, explicados a continuación.

6.1.2.1 Scripts

- **see_summary.py**

  Fichero que contiene el código a ejecutar en la consola GRR, gracias al cuál vamos a poder volcar los registros de logs al fichero de texto audits_grr.

  Su ruta completa es "~/grr_support/grr_summary/py_summary/see_summary.py". Puede verse su contenido en el Anexo H.

- **class_summary.py**

  Fichero que ejecutará en la consola de GRR el fichero anterior. Además, también incluye los métodos para realizar un registro de logs de esta aplicación.

  Su ruta completa es "~/grr_support/grr_summary/py_summary/class_summary.py". Puede verse su contenido en el Anexo G.

6.1.2.2 Logs

- **audits_grr**

  Fichero que contendrá todos los logs de GRR.

- **grr_logs**

  Fichero de logs de la aplicación.

6.1.3 Aplicación de bajo nivel

Esta herramienta no puede ser usada desde la web del Manager, sino que se accede a través de comandos desde el servidor GRR.

6.2 Inicio/Parada del servidor GRR: rbSt

6.2.1 Funcionamiento

Es la aplicación encargada de controlar el inicio, parada o reinicio del servidor GRR. En segundo caso no funcionaría ninguna de las aplicaciones desarrolladas con GRR vistas hasta ahora. Simplemente hace falta ejecutar el script shell_helpers[47] junto con un parámetro que le indique si queremos iniciar (grr_start_all), parar (grr_stop_all) o reiniciar (grr_restart_all) el servicio GRR.

En modo usuario (lo veremos en un capítulo posterior) simplemente pulsamos sobre los botones habilitados para ello desde la web del Manager.

6.2.2 Estructura

Está compuesto por solo un fichero: shell_helpers.sh[48]. Éste se encarga de realizar la acción que le indiquemos[49] en la ejecución del script. En el Anexo I puede verse su contenido.

[47] Explicado a en el apartado 6.2.2
[48] Este fichero ha sido modificado por el alumno para adaptarlo a sus necesidades. Originariamente ha sido creado por el equipo de GRR.
[49] Como parámetro.

7.DESPLIEGUE

"Cualquier tecnología suficientemente avanzada
es indistinguible de la magia"
- Arthur C. Clarke-

En este capítulo veremos los pasos necesarios para realizar el despliegue del proyecto en un entorno de trabajo.

7.1 Máquina virtual

Como ya comentamos en el capítulo 2, es necesario el uso de una máquina virtual para contener en ella el servidor de GRR, con SO Ubuntu Server 14.04. Ésta será ubicada en uno de los nodos del Manager de redBorder. Sin ella no podemos comenzar el proyecto, ya que hacemos uso de GRR para lanzar las peticiones y realizar la contención de EndPoints.

7.1.1 GRR

Una vez que tengamos la VM será necesario que instalemos el servidor de GRR en ella. Para ello tendremos que ejecutar las siguientes instrucciones y seguir los pasos que nos indiquen en pantalla.

Comandos 7.1 Instalación de GRR en la VM

```
cd /usr/bin
wget https://raw.githubusercontent.com/google/grr/master/scripts/install_sc
ript_ubuntu.sh
sudo bash install_script_ubuntu.sh
```

7.2 Dependencias necesarias para el proyecto

Dentro de la máquina virtual será necesario disponer del siguiente software:

- Dependencia de Python 2.7
- Kafka
- Aerospike
- S3

En caso de no encontrarse alguna de ellas (o ninguna) será necesario proceder a su instalación. Respectivamente, estas serían las órdenes ejecutadas para ello:

Comandos 7.2 Instalación de las dependencias del proyecto

```
$ sudo yum install python-devel
pip install kafka-python
pip install aerospike
sudo pip install boto
```

El equipo de sistemas de redBorder ha realizado una imagen de la máquina virtual en la que ya se encuentra GRR y todas estas dependencias.

7.3 rbChanges

Esta aplicación no se encuentra en la imagen de la máquina virtual. Es por ello que será necesario descargarse el siguiente software en función del entorno que se quiera montar:

Comandos 7.3 Descarga del proyecto desde gitlab

```
https://gitlab.redborder.lan/core-developers/grr_support/tree/release
```

El equipo de sistemas de redBorder ha elaborado una serie de scripts con los que convierten el software rbChanges en un servicio. Es por ello que será necesario que el paquete descargado de gitlab sea movido hasta la ruta **/root/grr_support.git/.**

En caso de tener que modificar algún parámetro ubicado dentro del fichero de configuración de rbChanges[50] se recomienda realizarlo antes de ejecutar la herramienta. Éste se encuentra ubicado en la ruta **grr_support.git/grr_service/scripts/config/parameters.json.** Si desea ver un ejemplo de este fichero puede visitar el Anexo D.

Para ejecutar el servicio de rbChanges tendríamos que ejecutar lo siguiente:

[50] Para obtener más información acerca de este fichero puede ir al apartado 4.4.1 del capítulo 4.

Comandos 7.4 Ejecución del software rbChanges

```
/etc/init.t/grrstates start
```

Se recomienda a partir de ahora (para el usuario encargado del mantenimiento de la herramienta) la visualización de los logs de rbChanges por si se llegara a detectar alguna anomalía en el servicio. Para la supervisión de logs, kafka, aerospike y s3 en una misma pantalla el alumno ha usado en su entorno de trabajo la herramienta Terminator. Se aconseja el uso de la misma o, en su defecto, una de las mismas características para comprobar con mayor comodidad que todo se va desarrollando con normalidad.

A continuación se muestra el comando usado para visualizar, en tiempo real de escritura, las últimas 100 líneas del fichero de logs (este número puede variar según las preferencias de cada usuario).

Comandos 7.5 Visualización del fichero de logs de rbChanges

```
tail -100f /var/log/grrstates/status_changes.log
```

Si por cualquier motivo fuera necesario modificar el fichero de configuración una vez arrancado rbChanges, tras realizar la modificación del mismo habría que reiniciar la aplicación:

Comandos 7.6 Reinicio de rbChanges

```
/usr/share/grr/scripts/shell_helpers.sh grr_restart_all
```

7.4 rbContention

Para poder hacer uso de esta herramienta será necesario que los python_hacks encargados del bloqueo y desbloqueo de los EndPoints sean almacenados en la ruta **aff4:/config/python_hack** del servidor GRR. Si se desea la visualización de éstos puede visitar el Anexo E y F.
En estos momentos se encuentran en la ruta **grr_support.git/python_hacks/firewall/.** Para que puedan ser usados por el flow ExecutePythonHack deberemos ejecutar el comando `grr_config_updater` de GRR, encargado de actualizar diversas características de la configuración de éste:

Comandos 7.7 Subida de python_hacks al repositorio del servidor GRR

```
/usr/bin/grr_config_updater upload_python --file="~/grr_support.git/
python_hacks/firewall/block_endpoint.py" --platform="windows" --arch amd64
```

```
/usr/bin/grr_config_updater upload_python --file="~/grr_support.git/
python_hacks/firewall/unblock_endpoint.py" --platform="windows" --arch amd64
```

Podemos ver que, a parte de la ruta actual de los python_hacks, también le indicamos para qué plataformas y arquitectura será utilizado.
Una vez hecho esto ya podemos hacer uso, tanto en la web de redBorder como en la CLI Console de GRR, de la contención de EndPoints. En el apartado 8.10 del siguiente capítulo veremos el uso de esta herramienta desde la interfaz gráfica, así que mostraremos aquí un ejemplo de su uso a bajo nivel. El único requisito para poder lanzar este flow es que seamos superusurio del sistema:

Comandos 7.8 Envío del flow ExecutePythonHack desde la CLI Console de GRR

```
sudo /usr/bin/grr_console --code_to_execute
'flow.GRRFlow.StartFlow(client_id="C.1921651fb48581d6",
flow_name="ExecutePythonHack", hack_name="~/grr_support.git/python_hacks/
firewall/block_endpoint.py")'
```

Hacemos uso del comando `grr_console`, encargado de lanzar las peticiones que le indiquemos, junto con el parámetro `code_to_execute`. Con este conseguimos que, sin abrir la consola de GRR, sean ejecutadas las órdenes que le pasemos. En este caso se ejecutará el lanzamiento de este flow para el EndPoint cuyo identificador GRR sea el indicado en 'client_id'. Para desbloquearlo haríamos exactamente lo mismo, pero en lugar de hacer referencia al fichero block_endpoint.py en el parámetro 'hack_name' se la haríamos a unblock_endpoint.py.

7.5 rbAudit

Esta herramienta sólo puede ser usada desde un terminal. Para su ejecución deberemos lanzar el siguiente comando:

Comandos 7.9 Ejecución de la auditoría mediante la terminal

```
cd ~/grr_support.git/grr_summary/
python class_summary.py
```

En el apartado 8.8 del siguiente capítulo podrá tener más información de esta aplicación.

7.6 rbSt

Esta aplicación puede ser usada tanto desde la web del Manager de redBorder como a bajo nivel en un terminal.

En el apartado 8.13 del siguiente capítulo mostraremos su uso desde la interfaz gráfica, así que ahora

mostraremos su ejecución a través de comandos. A continuación, podemos ver las órdenes necesarias para comenzar, parar y reiniciar el servicio GRR, respectivamente:

Comandos 7.10 Parada y arranque del servicio GRR a través de la terminal

```
/usr/share/grr/scripts/shell_helpers.sh grr_start_all

/usr/share/grr/scripts/shell_helpers.sh grr_stop_all

/usr/share/grr/scripts/shell_helpers.sh grr_restart_all
```

Si desea ver el código de esta herramienta puede visitar el Anexo I.

8.PRUEBAS Y TROUBLESHOOTING

"La vida no es un problema para ser resuelto,
es un misterio para ser vivido"
- Anónimo -

En todo proyecto es necesario, una vez terminadas e integradas todas las funcionalidades del mismo, establecer y llevar a cabo una batería de pruebas con las que corroborar su perfecta ejecución. Esto es precisamente lo que vamos a presentar en este capítulo. Además, también afrontaremos los diversos fallos y errores que podemos encontrar en las diversas pruebas. Con ellos acompañaremos su correspondiente solución para solventarlos y que sigan funcionando a la perfección.

Antes de empezar dejaremos constancia que en el Anexo K se encuentran explicadas, aunque no sea objeto del proyecto, las distintas vistas web que el equipo de redBorder ha realizado para incluir este proyecto en redBorder Malware. A partir del apartado 8.9 (y hasta el final del capítulo) se mostrarán las mismas para entender mejor la realización de las pruebas explicadas. Si no llegara a entender el funcionamiento de alguna de sus partes puede consultar dicho Anexo.

8.1 Desarrollo de las pruebas

Las pruebas han sido realizadas en el entorno de producción del proyecto redBorder Malware v1.0. Éste ha sido habilitado para que el cliente pueda ver los resultados del proyecto y empezar a familiarizarse con él, por lo que es muy importante que siempre esté operativo. Las pruebas se han llevado a cabo aquí para que, en caso de encontrar algún bug, seamos nosotros los primeros en darnos cuenta y arreglarlo.

8.2 Prueba 1.- Servidor GRR corriendo

Para comprobar que el servicio GRR está en funcionamiento tenemos que irnos al nodo donde se encuentra la VM de GRR mediante `ssh`. Para ello primero tenemos que conectarnos a uno de los nodos del Manager del entorno de producción. Elegimos aquel con IP 10.0.106.60.

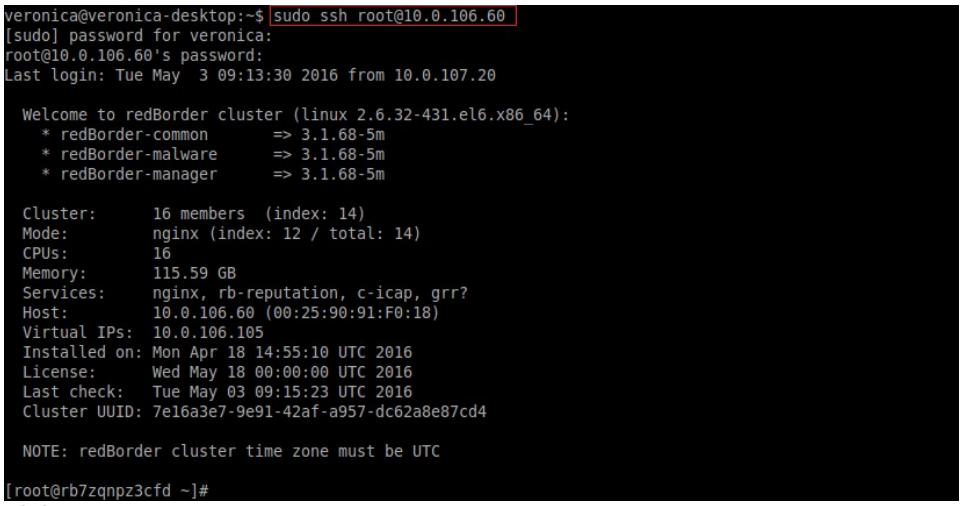

Figura 8.1 Conexión mediante ssh a uno de los nodos del Manager del entorno de producción

Tras esto miramos en qué nodo se encuentra la VM de GRR. Para ello hacemos uso del script `rb_get_services.sh` del Manager, el cual indica (para un servicio dado) en qué nodo/s se halla. El uso de éste requiere del parámetro `all`, con el que pedimos que nos muestre todos los nodos en los que se encuentre el servicio solicitado en la petición:

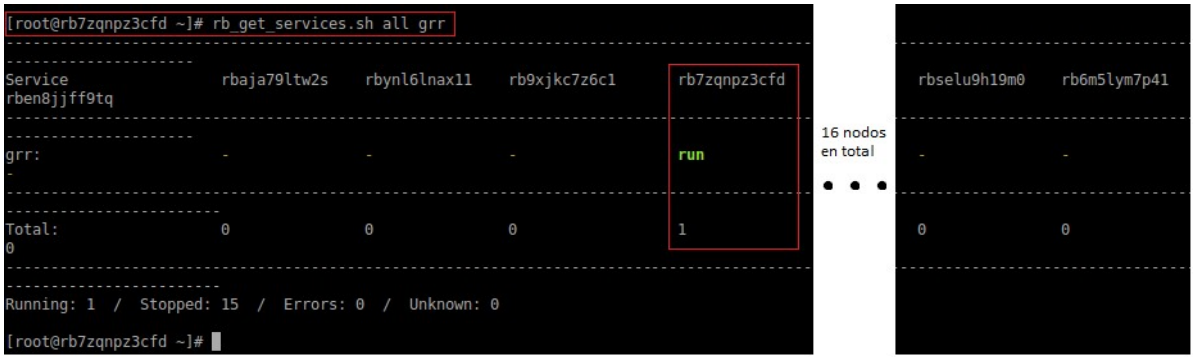

Figura 8.2 Nodo donde se encuentra la VM de GRR

Ahora ya es posible conectarnos al nodo donde está la VM de GRR mediante `ssh`:

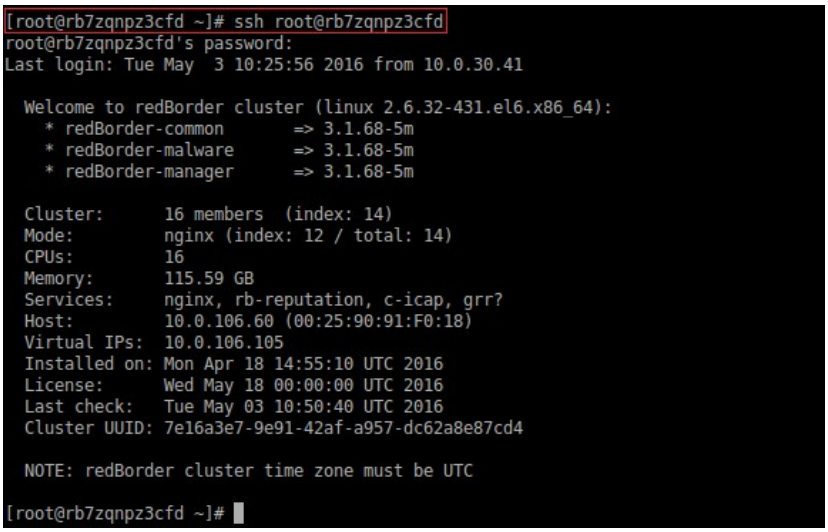

Figura 8.3 Conexión a la VM de GRR

Por último, para conectarnos a la VM de GRR, hacemos uso del comando `ifconfig` para obtener una lista con todas las interfaces de red ubicadas en el sistema y, así, ver cuál es la subred en la que se encuentra. En la lista resultante tenemos que buscar la interfaz grrbr0. Ésta nos muestra la dirección de subred privada 192.168.123.1/24, estando la VM en la primera dirección disponible, por lo que será la 192.168.123.2:

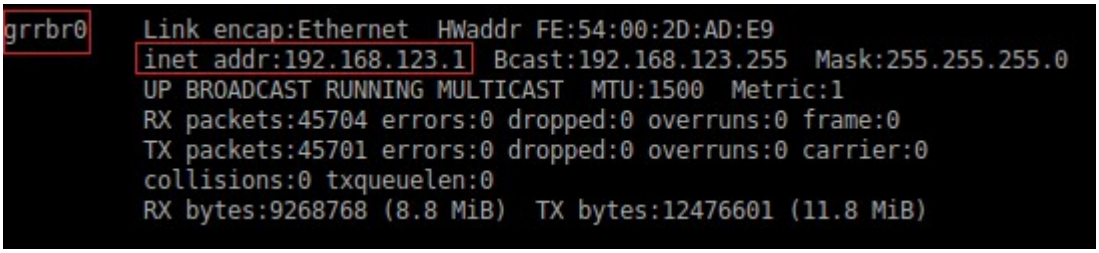

Figura 8.4 Interfaz grrbr0

Una vez que ya estamos en la VM de GRR comprobamos que esté corriendo el servicio GRR. Para ello hay que chequear que se estén ejecutando los procesos `start_http_server`, `start_ui` y `start_worker`. Esto lo logramos con el comando `ps aux`, visualizando así la información detallada de todos los procesos y, con la ayuda de `grep`, solo veremos aquella relacionada con el servicio GRR.

Figura 8.5 Comprobación de que el servicio GRR está corriendo

8.3 Prueba 2.- Servicio de Samza corriendo en el Manager

Este servicio es necesario que se esté ejecutando para que aparezcan en la web del Manager los cambios de estado detectados en la aplicación rbChanges. Para ello hacemos uso del script `rb_samza.sh` con el parámetro `-l`. Con éste listamos las tareas en ejecución que hacen uso de este servicio:

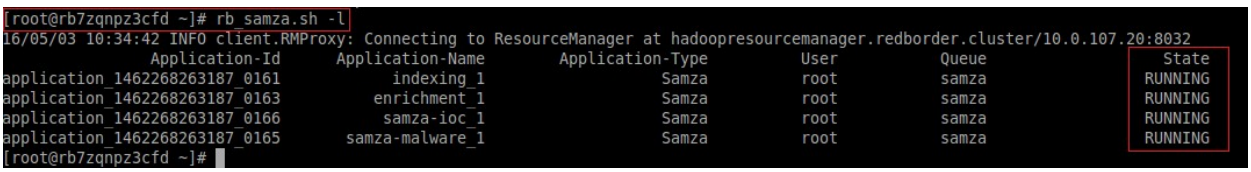

Figura 8.6 Comprobación de que Samza está corriendo

8.4 Prueba 3.- Servicio grrstates corriendo en la VM de GRR

Una vez conectados con la VM de GRR (como vimos en el apartado 7.2) ejecutamos `/etc/init.d/grrstates status` para comprobar que está corriendo el servicio de cambio de estados.

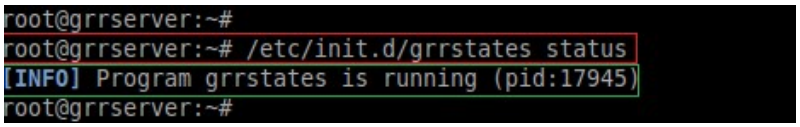

Figura 8.7 Comprobación de que el servicio grrstates esté corriendo

8.5 Prueba 4.- Correcto funcionamiento de kafka

Antes que nada, es necesario saber dónde se encuentra corriendo el servicio kafka, por lo que en uno de los nodos del Manager ejecutamos el script `rb_get_services.sh` haciendo referencia a éste:

Figura 8.8 Nodos en los que se encuentran el servicio kafka

Podemos ver que, para kafka, hay dos nodos en los que se está ejecutando este servicio. Elegimos uno de ellos para conectarnos.

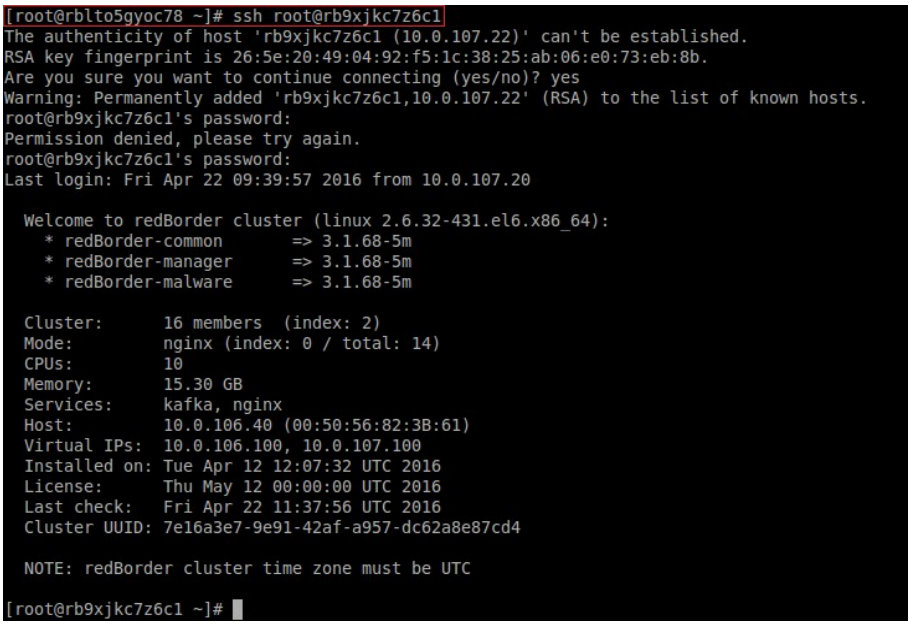

Figura 8.9 Conexión mediante ssh a uno de los nodos del Manager del entorno de producción en el que se encuentra kafka

Es necesaria para la realización de la prueba la apertura de dos terminales para ejecutar en una de ellas el productor y en la otra el consumidor de kafka, lanzando `rb_producer.sh` y

`rb_consumer.sh` respectivamente. En ambos es utilizado el parámetro `-t`, con el que indicamos que el topic para el envío y recepción de mensajes será `rb_changes`**:**

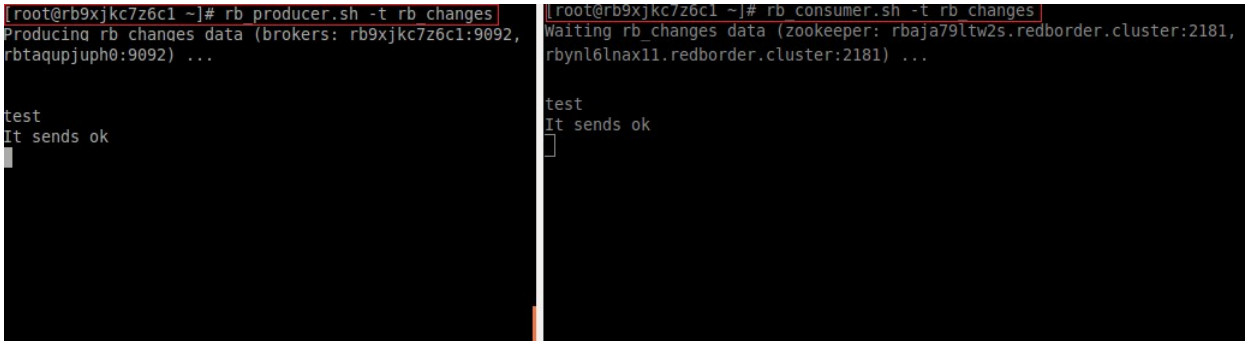

Figura 8.10 Comprobación de la recepción de los mensajes en el consumidor enviados desde el productor

Podemos ver que los mensajes enviados desde el productor llegan correctamente al consumidor.

8.6 Prueba 5.- Correcto funcionamiento de S3

Para ello es necesario que veamos primero los nodos donde se encuentra Riak, lanzando para ello `rb_get_services.sh`.

Figura 8.11 Nodos en los que se encuentran el servicio kafka

Una vez obtenidos elegimos uno de ellos para conectarnos mediante `ssh` (por ejemplo, aquel de nombre 'rbspzxodon4s').

A continuación, verificamos que coinciden correctamente tanto el `access_key` como el `secret_key` de S3 con lo establecido en los parámetros de configuración del programa rbChanges. Para ello, en el nodo donde se encuentra Riak, abrimos el fichero s3cfg-malware ayudándonos del editor de texto `vim` y, abriendo otro terminal con conexión sobre la VM de GRR, comparamos con el fichero de configuración[51] del software.

Podemos ver que todo está correcto (sólo mostraremos un trozo del fichero s3cfg-malware con los datos que nos interesan para esta parte):

[51] Este fichero de configuración está ubicado en **~/grr_support.git/grr_service/scripts/config/parameters.json.**

```
[default]
access_key = XZPT6XGOQQCAJGRAHSHL
secret_key = cGztZoG2o92gfUsFWUdi4rFeCNXzIBIrAQIORw==
host_base = s3.redborder.cluster
bucket_location = US
host_bucket = %(bucket)s.s3.redborder.cluster
```

Figura 8.12 Trozo del archivo s3cfg-malware

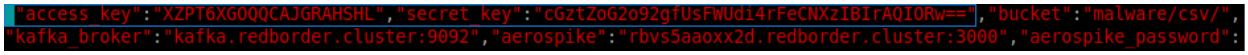

Figura 8.13 Fichero parameters de rbChanges

Si queremos ver que de verdad se encuentran los ficheros CSV en S3 ejecutamos `s3cmd -c .s3cfg-malware ls s3://malware/csv/.` Éste nos mostrará mediante el `ls` la ruta[52] donde se encuentran estos archivos en S3.

Mostraremos a continuación sólo un trozo de la salida que ha generado, debido a que contiene una gran cantidad de ficheros en este momento:

```
[root@rbspzxodon4s ~]# s3cmd -c .s3cfg-malware ls s3://malware/csv/
2016-04-21 15:28     26436   s3://malware/csv/C.2ead5024c7a8d529_02bf4e9051027048a3d2ef1bab156ec5.csv
2016-04-20 11:47      9830   s3://malware/csv/C.2ead5024c7a8d529_0330740511ab443fcee66de49ab13acc.csv
2016-04-21 20:08     26425   s3://malware/csv/C.2ead5024c7a8d529_08741403539651f9280bd650007bc50f.csv
2016-04-20 11:26     19819   s3://malware/csv/C.2ead5024c7a8d529_0dbcfad4a55badf6e8cabdca61e5ebe5.csv
2016-04-22 04:54     26173   s3://malware/csv/C.2ead5024c7a8d529_0e846415fe0b5212ab1f5ef4953265e4.csv
2016-04-21 21:38     10079   s3://malware/csv/C.2ead5024c7a8d529_10378813b57469ca6ba1aa0a08e0e3a0.csv
2016-04-22 00:14     26433   s3://malware/csv/C.2ead5024c7a8d529_151d66ebcbec462d9ea4a85666fd6da4.csv
2016-04-22 05:48     10071   s3://malware/csv/C.2ead5024c7a8d529_168b83dd89c6304f56cce7fef909ed74.csv
2016-04-22 06:58     10337   s3://malware/csv/C.2ead5024c7a8d529_17d9ba026f8b724c408b4f663b73c2d5.csv
2016-04-22 03:43     26184   s3://malware/csv/C.2ead5024c7a8d529_1bd25779a2b81f7f6687712b442464a4.csv
2016-04-22 09:53      9806   s3://malware/csv/C.2ead5024c7a8d529_1d0cefba98335bf2aed2c2d805cab8c8.csv
2016-04-22 07:33     10096   s3://malware/csv/C.2ead5024c7a8d529_2122529a4748e7b6474eb513de58b090.csv
2016-04-22 10:28      9559   s3://malware/csv/C.2ead5024c7a8d529_222c0a56602e6eb420348d80e981b9b3.csv
2016-04-22 01:09     10130   s3://malware/csv/C.2ead5024c7a8d529_2de7c17712bd59589c3daf5c927f70f1.csv
2016-04-21 16:33     10104   s3://malware/csv/C.2ead5024c7a8d529_3883d230eda892be5dda3a99226189d4.csv
2016-04-21 19:32     26424   s3://malware/csv/C.2ead5024c7a8d529_3998c48b98d1dba2c7279cccc4e035e4.csv
```

Figura 8.14 Contenido de S3 para los ficheros CSV del programa StateChanges

Por último, también podemos comprobar que un fichero concreto se ha mandado a S3. Para ello ejecutamos `s3cmd -c .s3cfg-malware ls s3://malware/csv/ | grep` acompañado del nombre del CSV (podemos obtenerlo del mensaje kafka[53], p.e). Un ejemplo de ello sería:

```
s3cmd -c .s3cfg-malware ls s3://malware/csv/ | grep
C.43a4d19ed4ec2e82_75e88de9a8b30618cd8e5d41f4ba3aec.csv
```

8.7 Prueba 6.- Funcionamiento de Aerospike

Antes que nada, es necesario saber dónde se encuentra corriendo el servicio Aerospike, por lo que en uno de los nodos del Manager ejecutamos el script `rb_get_services.sh` haciendo referencia a este servicio.

[52] Esta ruta es **/malware/csv**
[53] Ver figura 4.9 del capítulo 4.

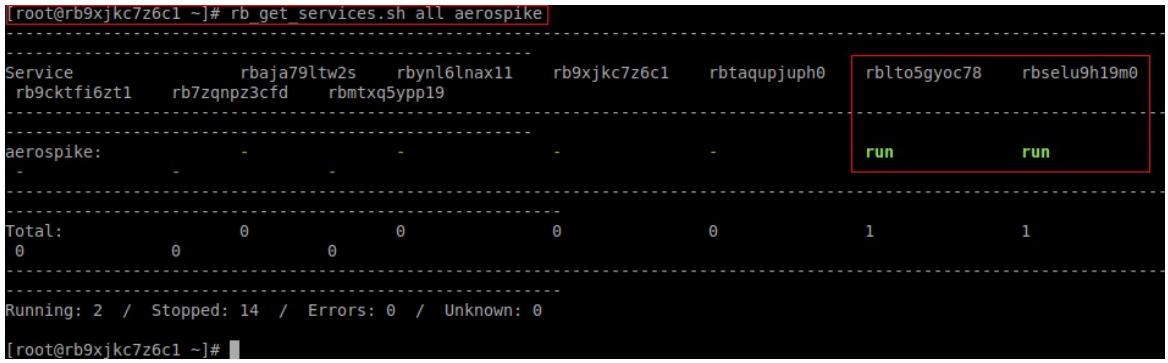

Figura 8.15 Parte de los nodos en los que se encuentra el servicio Aerospike

Ahora nos conectamos a uno de los nodos en los que se encuentre el servicio mediante `ssh` (p.e al nodo 'rblto5gyoc78').

Una vez que hemos realizado la conexión accedemos a Aerospike ejecutando el comando `aql` y adjuntándole el nombre de la máquina.

Figura 8.16 Tabla con la que trabajaremos en Aerospike

Con esto comprobamos que podemos acceder al servicio correctamente, la base de datos está en funcionamiento y la tabla donde se van a registrar los ficheros CSV existe y, además, hay dos entradas por EndPoint, es decir, una por cada flow (Netstat y ListProcesses).

En el caso de que en algún momento rbChanges tuviera problemas de conexión con Aerospike, podría ser que este servicio haya sido cambiado de nodo por algún motivo. Esto es debido a que en el fichero de configuración del nodo donde se encuentra GRR se indica la máquina donde está la BBDD. Si no se ha modificado con los datos del nuevo nodo puede dar estos problemas.

Dejamos a continuación un ejemplo de este fichero de configuración.

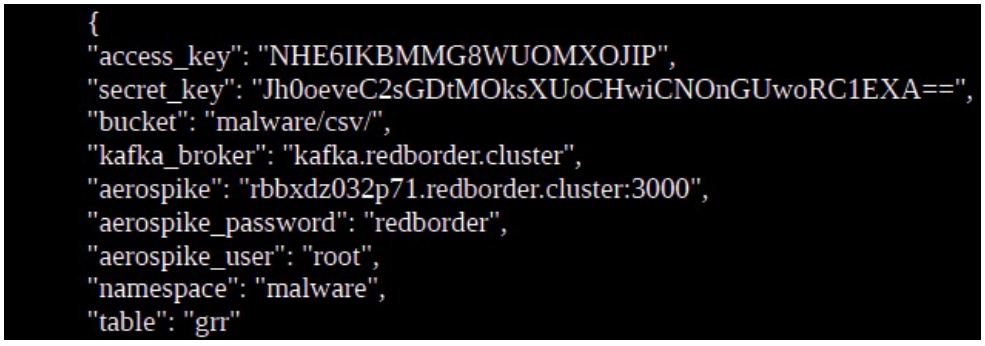

Figura 8.17 Fichero configuración del nodo de GRR

8.8 Prueba 7.- Auditoría de GRR

Primero nos conectamos a la MV de GRR y ejecutamos `cd ~/grr_support.git/grr_summary` para irnos hasta esa carpeta. Una vez ahí nos fijamos si sólo aparece el fichero class_summary y el directorio py_summary. Si es así es porque no se ha ejecutado todavía.

Para la ejecución de la herramienta hacemos `python class_summary.py`. Una vez hecho nos aparecerá el directorio de logs y el fichero con los resultados de la auditoría (audits_grr).

```
root@redBorder-GRR-server:~/grr_support.git/grr_summary# ls
audits_grr.txt  class_summary.py  logs  py_summary
root@redBorder-GRR-server:~/grr_support.git/grr_summary#
```

Figura 8.18 Directorio grr_summary tras ejecución de class_summary.py

Por último, comprobamos que la escritura en el fichero audits_grr y en el fichero grr_logs funciona correctamente.

```
root@redBorder-GRR-server:~/grr_support.git/grr_summary/logs# cat grr_logs.txt
Fri Apr 22 07:20:52 2016 Class Summary has just initialized succesfully
Fri Apr 22 07:20:55 2016 It have just obtain GRR's behavior
Fri Apr 22 07:37:46 2016 Class Summary has just initialized succesfully
Fri Apr 22 07:37:48 2016 It have just obtain GRR's behavior
root@redBorder-GRR-server:~/grr_support.git/grr_summary/logs#
```

Figura 8.19 Extracto del fichero de logs de la auditoría grr_logs.txt

```
message AuditEvent {
 action : HUNT_STARTED
 description : u'rbstatus Cron'
 flow_name : u''
 id : 1392
 timestamp : RDFDatetime:
    2016-04-22 12:13:07
 urn : RDFURN:
    aff4:/hunts/H:6AC9A789
 user : u'GRRWorker'
}
message AuditEvent {
 action : RUN_FLOW
 flow_name : u'PackedVersionedCollectionCompactor'
 id : 1393
 timestamp : RDFDatetime:
    2016-04-22 12:18:07
 urn : RDFURN:
    aff4:/cron/PackedVersionedCollectionCompactor/F:209846C9
 user : u'GRRCron'
}
```

Figura 8.20 Extracto del fichero audits_grr.txt

8.9 Prueba 8.- Registro de un nuevo EndPoint

Esta prueba consiste en registrar un nuevo EndPoint y comprobar que queda registrado en el servidor GRR y se visualiza correctamente desde la web. Esto último puede tardar algunos minutos.

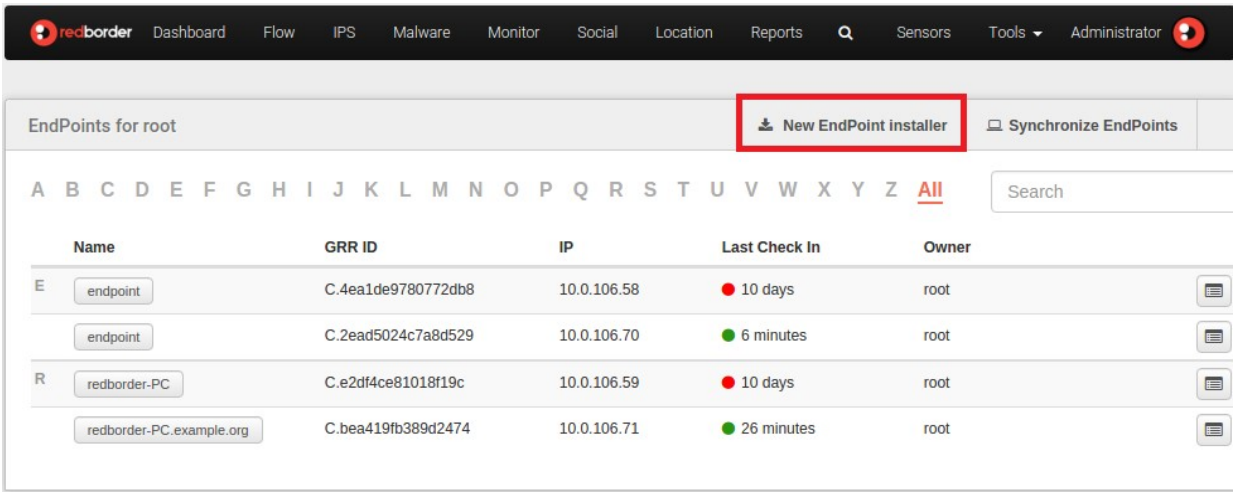

Figura 8.21 Vista de EndPoints en la web del Manager

Desde la vista de EndPoints tenemos la opción de descarga del instalador. En él está incluido el agente de GRR que se instalará en el equipo final que se lo descargue. Este instalador, cuyo formato es .zip, está disponible tanto para SO Windows de 32 y 64 bits.

Una vez ejecutado el instalador en el EndPoint debería aparecer en esta vista web. Esta acción puede tardar entre 1 y 10 minutos.

En caso de que transcurra un tiempo y siga sin aparecer tenemos la posibilidad de pulsar el botón de sincronización de los EndPoints. Éste se encarga de hacer que GRR se comunique con todos los agentes de los EndPoints. Al obtener respuesta comprueba los que están activos, por lo que al recargar la página debería aparecer.

8.10 Prueba 9.- Contención

Para proceder al bloqueo de conexiones entrantes y salientes de un EndPoint accedemos a uno en particular desde la web del Manager. Una vez en la página de información del mismo pulsamos sobre "Block EndPoints". En la siguiente imagen podemos observar, además, cómo el estado de bloqueo nos indica que el equipo final está desbloqueado en ese momento, es decir, no tiene aplicada la herramienta de contención.

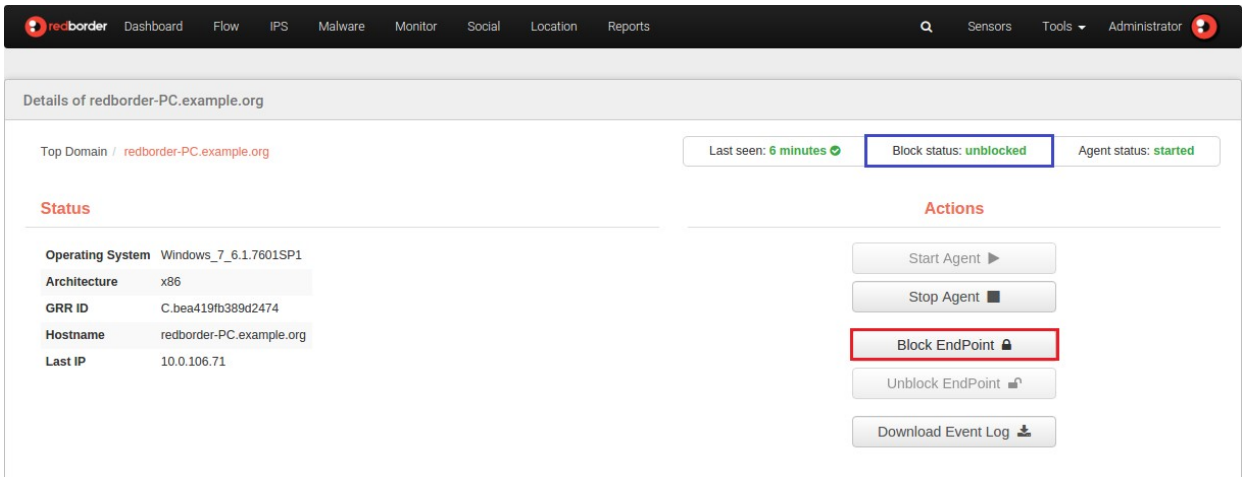

Figura 8.22 Bloqueo de EndPoint desde la web del Manager

A continuación nos aparecerá una ventana emergente en la que deberemos indicar las direcciones IP de aquellos equipos con los que queramos seguir manteniendo el contacto. Aquí no será necesario introducir la IP del servidor GRR, ya está indicado internamente.
En este caso, p.e, indicamos que queremos seguir tener comunicación con la IP "8.8.8.8".

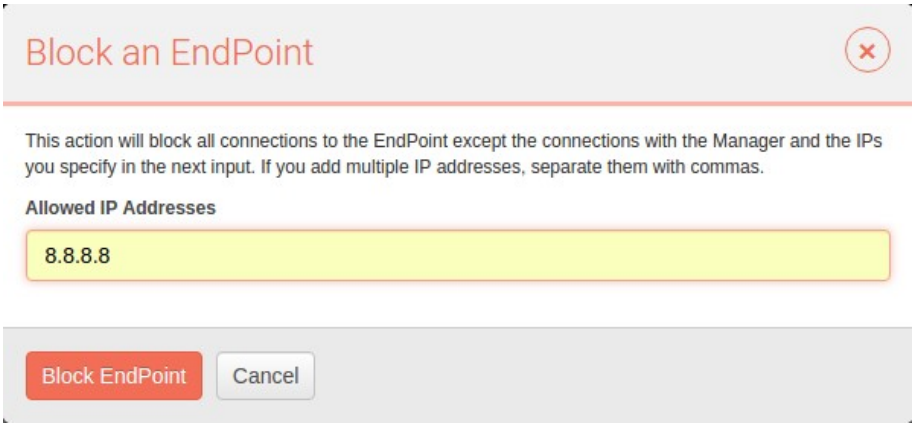

Figura 8.23 Introducción de las IP con las que mantener el contacto en la contención desde la web

Podemos ver en la siguiente imagen que, tras aplicar la contención, el estado de bloqueo ha cambiado, y que ahora la opción disponible es la de desbloqueo del EndPoint.

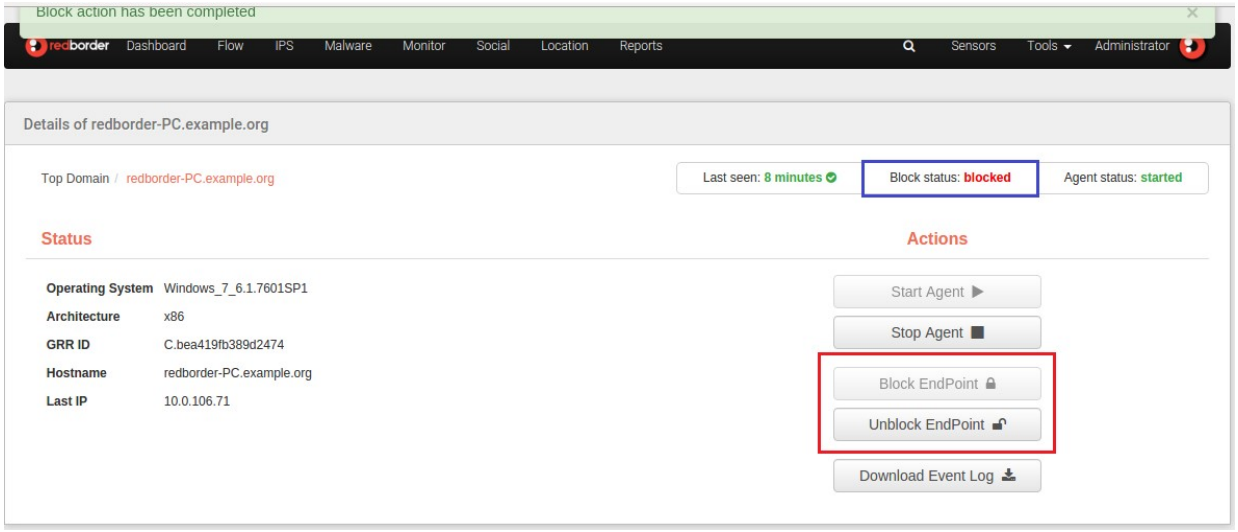

Figura 8.24 El estado de bloqueo ha sido modificado

Ahora nos vamos hasta el EndPoint. Probamos a hacer `ping` a las IPs 8.8.8.8 y a la del nodo donde se encuentra GRR, que en este caso es la 10.0.106.60. Deberíamos obtener respuesta de ambas. Comprobamos que es así efectivamente.

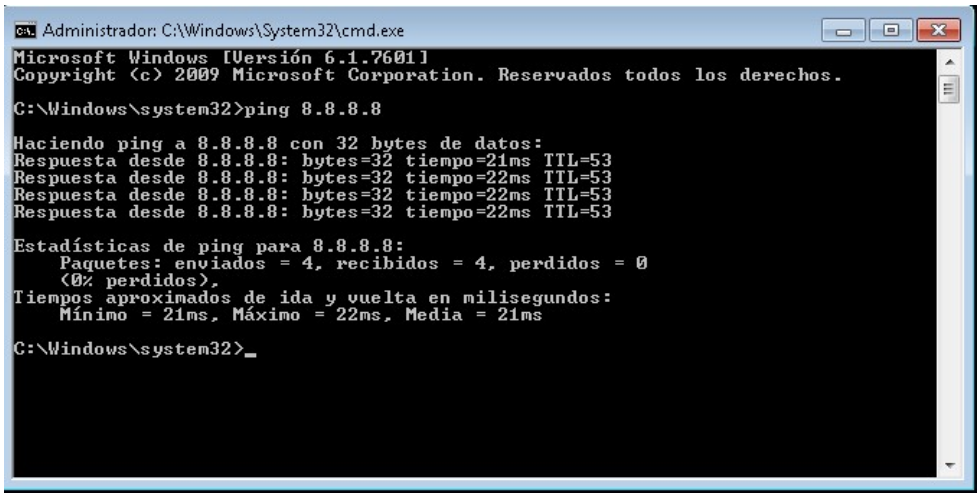

Figura 8.25 Ping a la IP 8.8.8.8

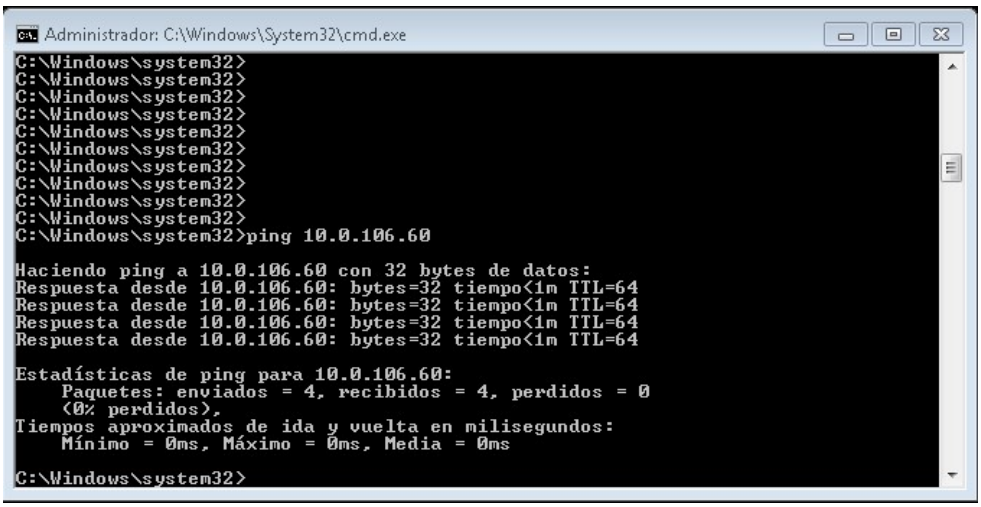

Figura 8.26 Ping al nodo donde se encuentra GRR

Ahora chequeamos como no podemos acceder a ninguna otra dirección IP.

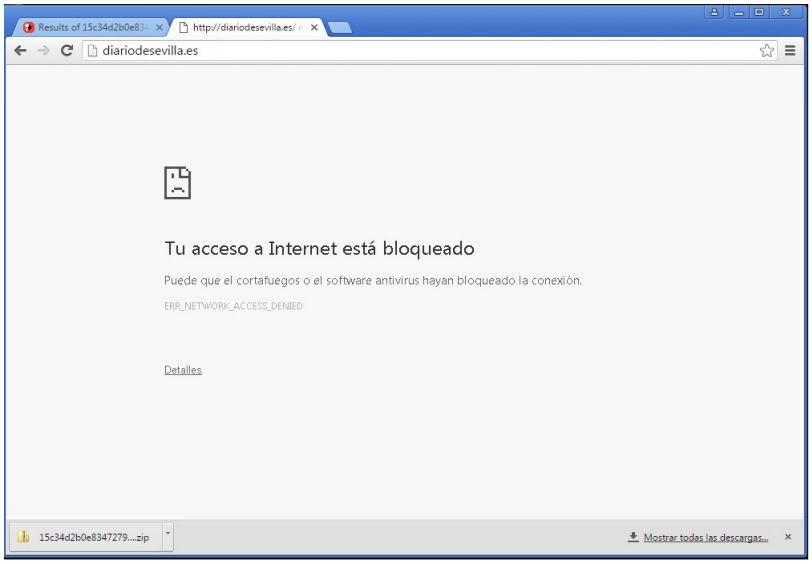

Figura 8.27 Acceso a "Diario de Sevilla" denegado

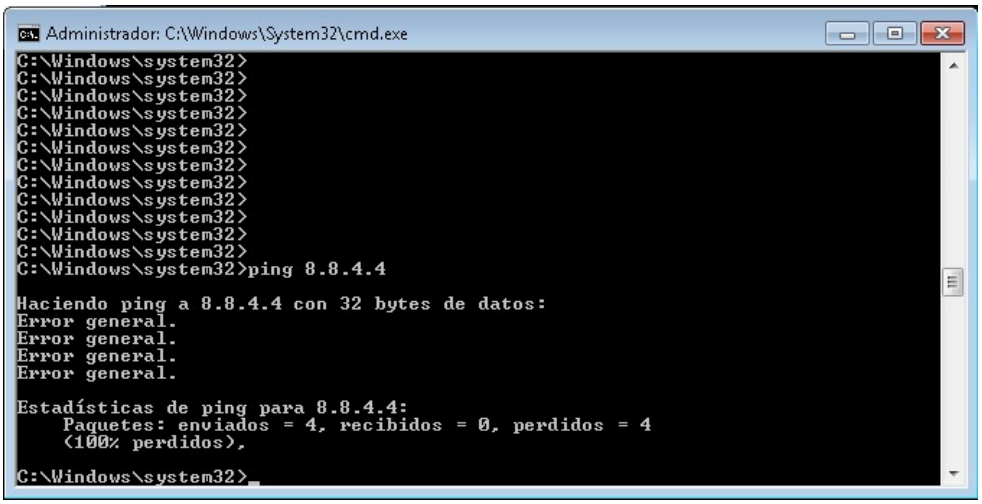

Figura 8.28 Acceso a la IP 8.8.4.4 denegado

Como ya vimos en la figura 7.22, luego de bloquear un EndPoint debe estar disponible la acción contraria para volver a dejarlo a su anterior estado. Comprobamos que ,tras pulsar "Unblock EndPoint", se cumple esto. Volverá a modificarse el estado de bloqueo en la web una vez realizada esta acción.

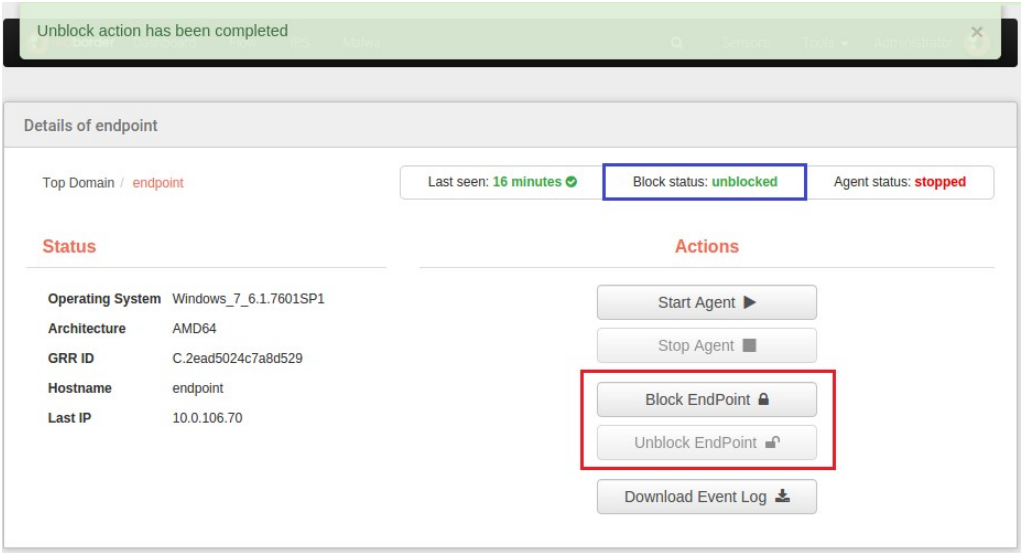

Figura 8.29 Bloqueo del EndPoint desde la web del Manager

Por último, comprobamos como ya volvemos a tener conexión con todos.

Figura 8.30 Acceso a "Diario de Sevilla" permitido

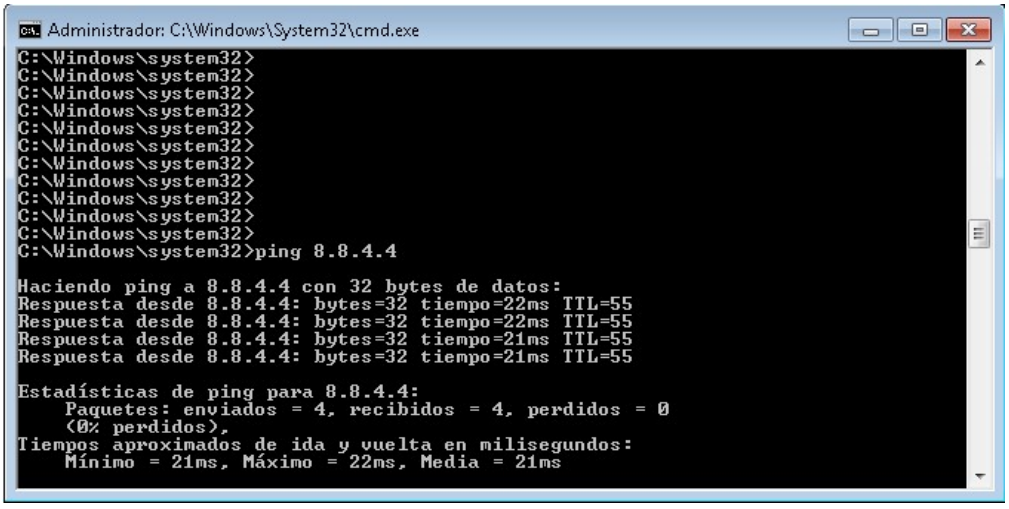

Figura 8.31 Acceso a la IP 8.8.4.4 permitido

En caso de que no llegara a funcionar la contención de los equipos finales podría deberse a que el servicio GRR no está funcionando (ver apartado 7.13) o que no están añadidos los python_hacks en el servidor.

Podemos comprobar esto último yéndonos a la ruta `/usr/lib/python2.7/dist-packages/grr/server/local/python_hacks/` y ver si aparecen los scripts block_endpoint y unblock_endpoint.

Si no se cumple ninguno de los dos casos anteriores, podemos ver los errores provocados por los python_hacks revisando uno de los ficheros de logs de GRR, encargado de listar los fallos de los flows lanzados:

```
tail -f -n 100 /var/log/grr/grr-worker.log
```

8.11 Prueba 10.- Listado de los EndPoints

En la lista donde se encuentran todos los EndPoints vinculados a GRR debe mostrarse, en la columna "Last Check In", el tiempo desde la última vez que se detectó que los equipos finales están en funcionamiento.

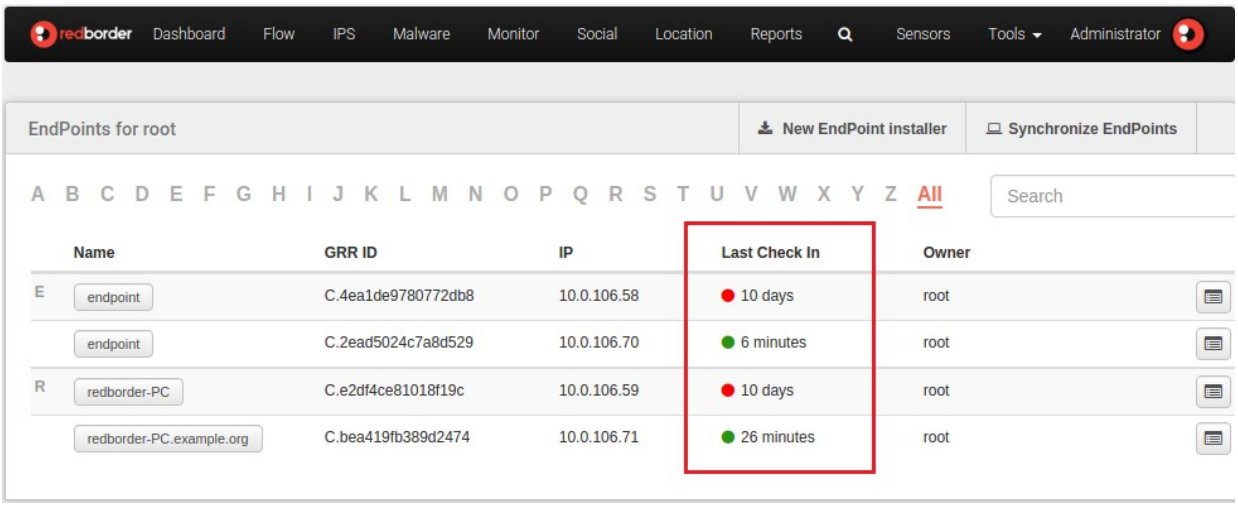

Figura 8.32 Aparición del tiempo desde el último chequeo con rbChanges en los EndPoints

GRR tiene monitorizado automáticamente, mediante crons, el envío periódico del flow Interrogate. Éste se encarga de comprobar que los EndPoints están activos. Por decirlo de alguna manera, es parecido al envío de un `ping`: si obtenemos respuesta es porque lo están. El equipo de redBorder encargado de la web se ha aprovechado de esto para indicarlo en la misma.

Las esferas rojas nos indican que el equipo final, cuyo identificador GRR se corresponde con la columna GRR ID, se encuentra en estos momentos apagado. Además, se le añade la última vez que se avistó actividad en él. Por el contrario, las de color verde nos hacen saber que los equipos están encendidos. Éstas se encuentran acompañadas de la última vez que el flow Interrogate fue ejecutado en los equipos.

En esta prueba tenemos que comprobar que los equipos apagados aparezcan referenciados con el símbolo rojo y los encendidos con el verde. En el caso de que haya algún error y la web muestre como 'apagado' algún EndPoint que no lo esté, puede deberse a que:

- El servicio GRR esté parado (ver apartado 7.13), por lo que el agente del equipo final no

puede recibir los flows[54].

- El EndPoint haya pasado a un estado “dormido” debido que está configurado con la opción de ahorro de energía.
- Hubo algún problema con Samza o Druid en el momento de registrar el Endpoint y la información registrada en el sistema es incorrecta.
- Porque esté a la espera de reiniciarse el equipo, p.e, para una actualización del sistema.

8.12 Prueba 11.- Malware Settings

Hay que comprobar que aparecen los servicios de GRR "grr-http-server", "grr-ui" y "grr-worker" corriendo en el sistema. Para ello nos fijamos en el parámetro “Status”. Si el estado de los tres aparece como “Running” está todo correcto. Además, deben indicar el PID cada uno de ellos.

También es importante que nos fijemos que se especifique la dirección IP del nodo donde se encuentra la VM de GRR y que ésta sea correcta.

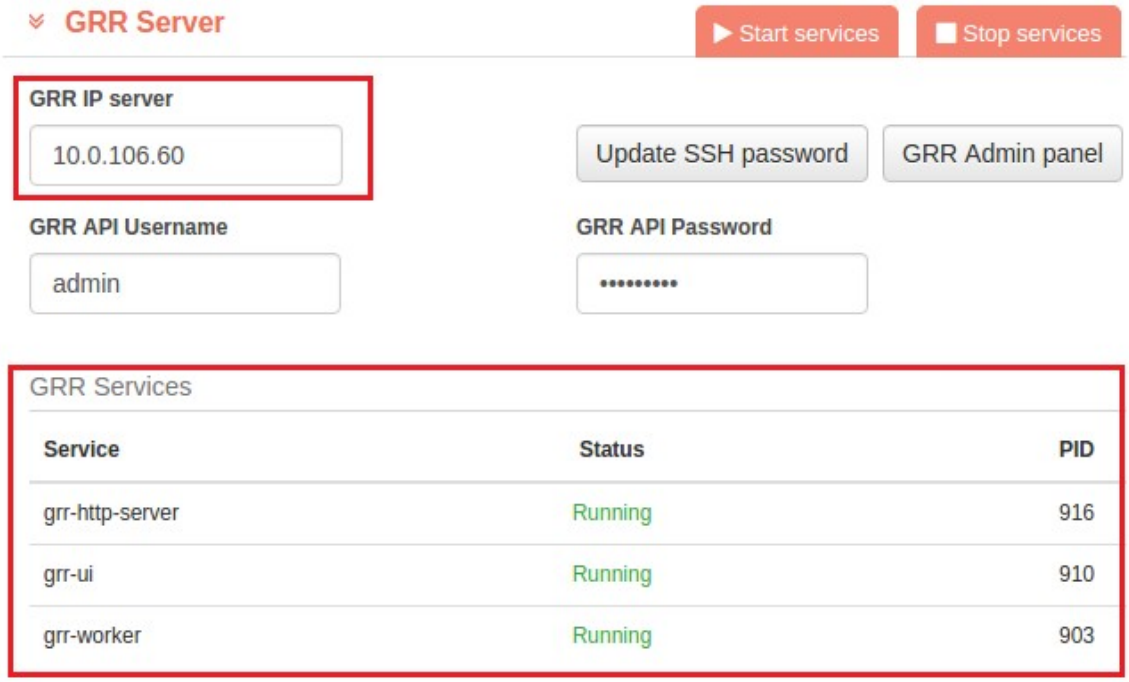

Figura 8.33 Comprobación de los servicios GRR en la ventana “Malware Settings”

8.13 Prueba 12.- Parada y arranque del servidor GRR

En esta prueba vamos a comprobar que el servicio GRR se para y reanuda correctamente. Comenzamos parando el servicio pulsando el botón "Stop services".

[54] Las peticiones enviadas se quedan en cola a la espera de que vuelva a estar operativo el agente cliente de GRR.

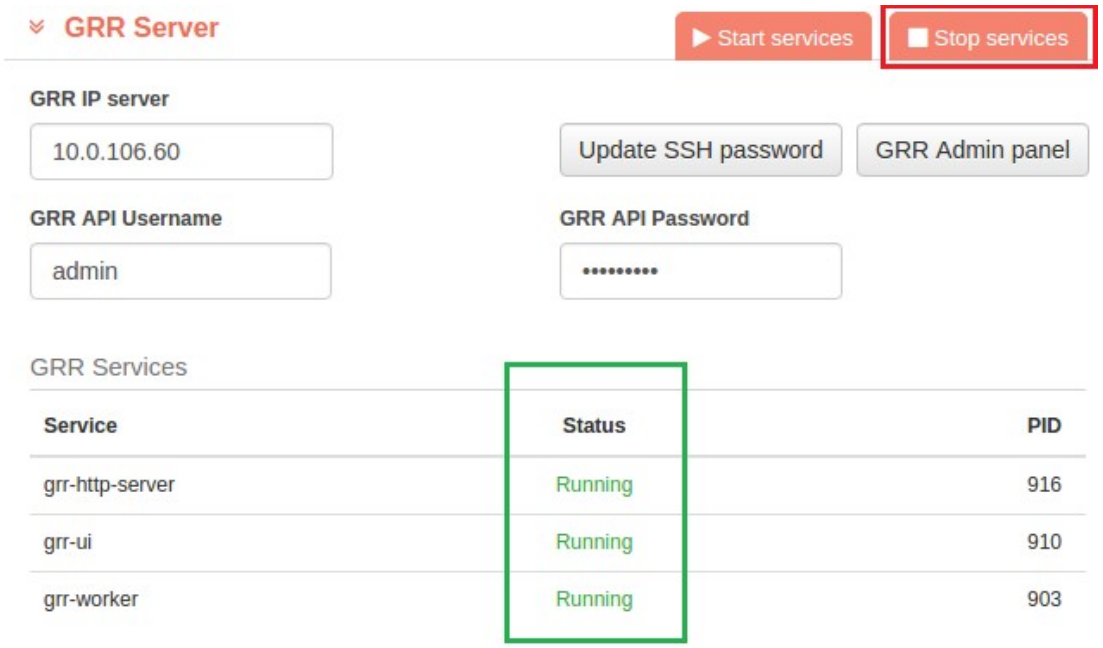

Figura 8.34 Parada del servicio GRR desde la web

Seguido de ello nos vamos a la MV de GRR y ejecutamos `ps aux | grep grr`. Corroboramos que no están corriendo los servicios de GRR.

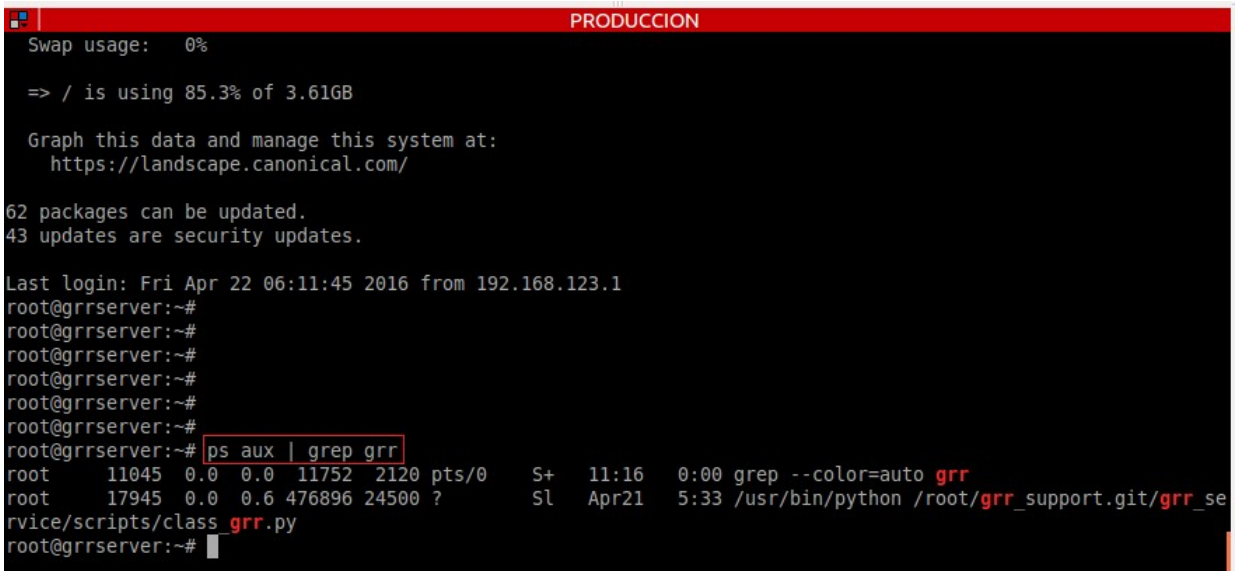

Figura 8.35 Comprobación de que los servicios de GRR no están en funcionamiento

También desde la web podemos ver que el servicio GRR no está arrancado. Para ello nos fijamos en los parámetros “Status” y “PID”.

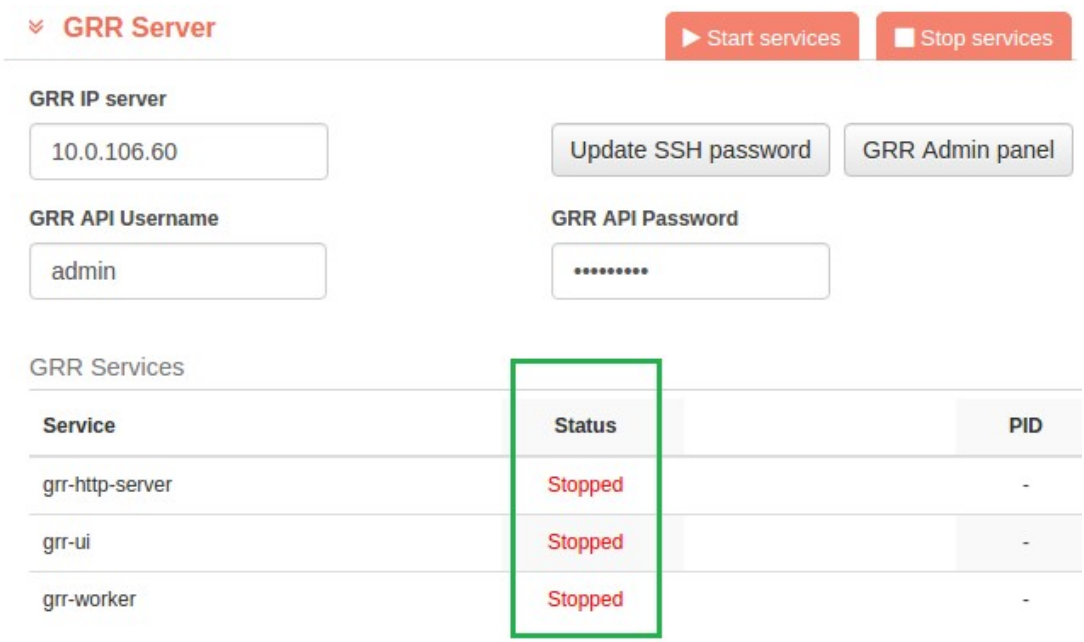

Figura 8.36 Reinicio del servicio GRR desde la web

Finalmente, para restaurar el servicio GRR, pulsamos el botón "Start services". Desde la consola podemos comprobar que se han ejecutado los cambios.

Figura 8.37 Comprobación de que los servicios de GRR están en funcionamiento

Si hay algo que está fallando en GRR y no sabemos qué puede ser podemos ver los logs de este servidor en la ruta `/var/log/grr`. En caso de seguir fallando podemos hacer uso de esta herramienta para ver si ,parando y reanudando GRR, llegara a funcionar de nuevo (sí, es una solución un poco rudimentaria, pero lo hacemos como último recurso).

Desde la consola también podemos reiniciar GRR ejecutando

```
/usr/share/grr/scripts/shell_helpers.sh grr_restart_all
```

ó

```
/usr/share/grr/scripts/shell_helpers.sh grr_stop_all
/usr/share/grr/scripts/shell_helpers.sh grr_start_all
```

El usuario también puede ver desde la web que el servicio GRR ha vuelto a restaurarse. Tendríamos una imagen igual que la figura 7.32.

8.14 Prueba 13.- Acceso a la web de GRR Server

En esta prueba vamos a comprobar que podemos acceder a la web del servidor GRR sin ningún

problema. Para ello presionamos sobre el botón "GRR Admin Panel". Es importante que la dirección IP del servidor GRR sea correcta, al igual que el nombre de usuario y la contraseña con la que se ha registrado el administrador en GRR.

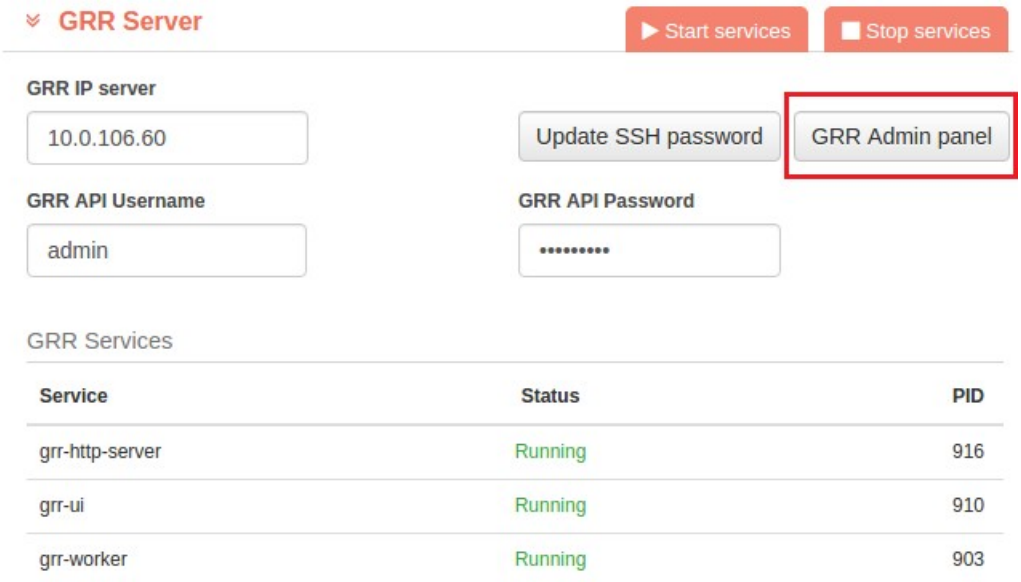

Figura 8.38 Acceso a la interfaz de GRR desde la web del Manager

Una vez pulsado este botón se debería abrir una nueva pestaña en el navegador con la web de administración del servidor GRR.

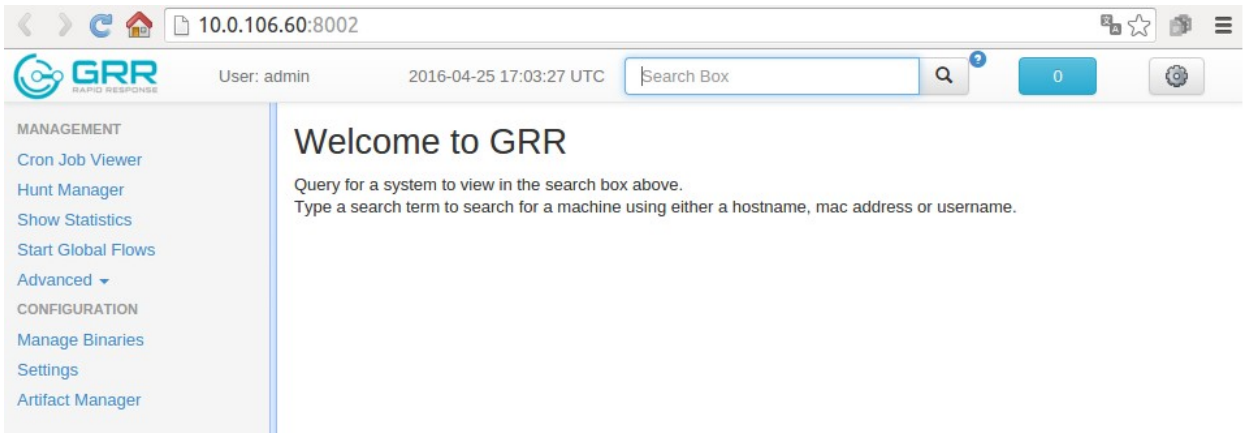

Figura 8.39 Interfaz web de GRR

8.15 Prueba 14.- Lista de cambio de estados correcta

En esta prueba comprobamos que no hay cambio de estados repetidos y que la paginación se realiza correctamente.

Cuando ocurre un cambio de estado es cuando aparecerá en la siguiente vista web, por lo que para una entrada dada nos descargaremos los dos CSV (old CSV y New CSV) y comprobaremos que es verdad que son diferentes.

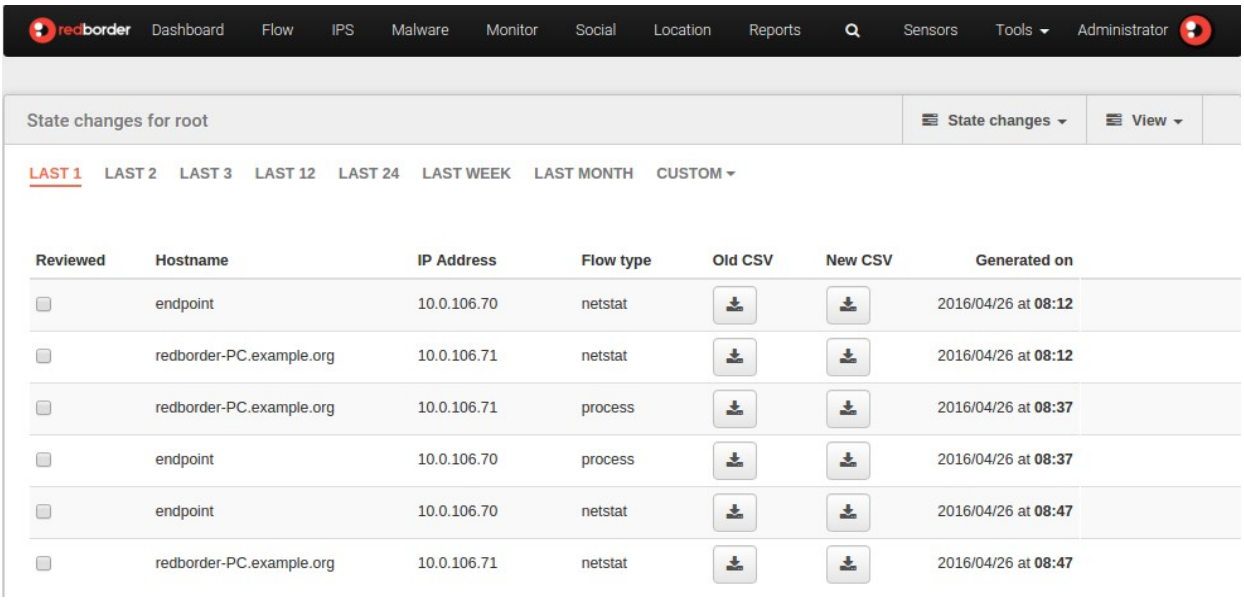

Figura 8.40 Vista de cambios de estado en la web del Manager

Si no se llegara a visualizar en esta vista algún cambio de estado que supiéramos que se ha producido (al verlo en un mensaje kafka) habría que revisar, a bajo nivel, si está corriendo el servicio Samza desde el Manager. En el caso de que comprobemos que no está en funcionamiento se debe ejecutar `rb_samza.sh -t indexing -ke`. Para lanzar todos los procesos necesarios de Samza realizamos la siguiente orden:

```
for i in indexing enrichment malware ioc; do rb_samza.sh -t $i -ke; done
```

Por último, comprobamos que todos esos procesos están corriendo ejecutando el comando `rb_samza.sh -l`.

En caso de que con Samza no hubiera ningún problema podría ser que no se estuvieran enviando los CSV a S3. Para comprobarlo nos fijamos cuál es el nombre del fichero en el mensaje kafka y lo bucamos en S3(ver apartado 7.6).

8.16 Prueba 15.- Filtrado de cambios de estado correcto

En esta prueba comprobaremos que en la vista de cambios de estado cada una de las entradas se puede marcar como “revisada”. Al recargar la web se deben guardan los cambios.

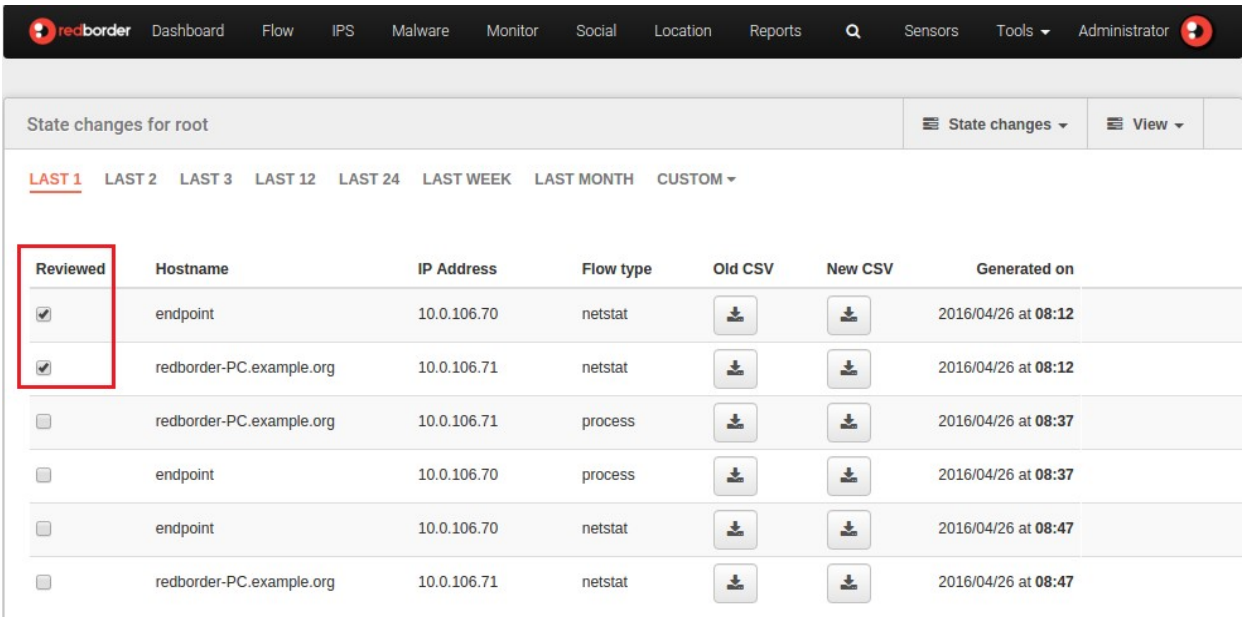

Figura 8.41 Vista de los cambios de estado con el parámetro 'revisado'

Tenemos la opción de marcar manualmente esos casilleros como ayuda para saber qué CSV son los que hemos revisado.

También, a través del desplegable "View", podemos filtrar y ver sólo las entradas revisadas ("show only reviewed elements"), las no revisadas ("show unreviewed elements") o todas ("show everything"). Comprobamos que, efectivamente, se muestra correctamente.

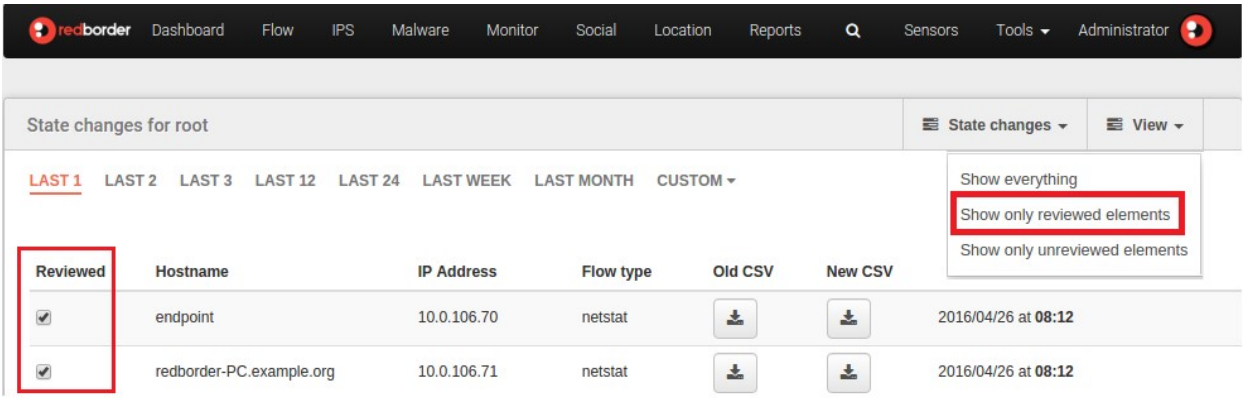

Figura 8.42 Vista de las entradas revisadas

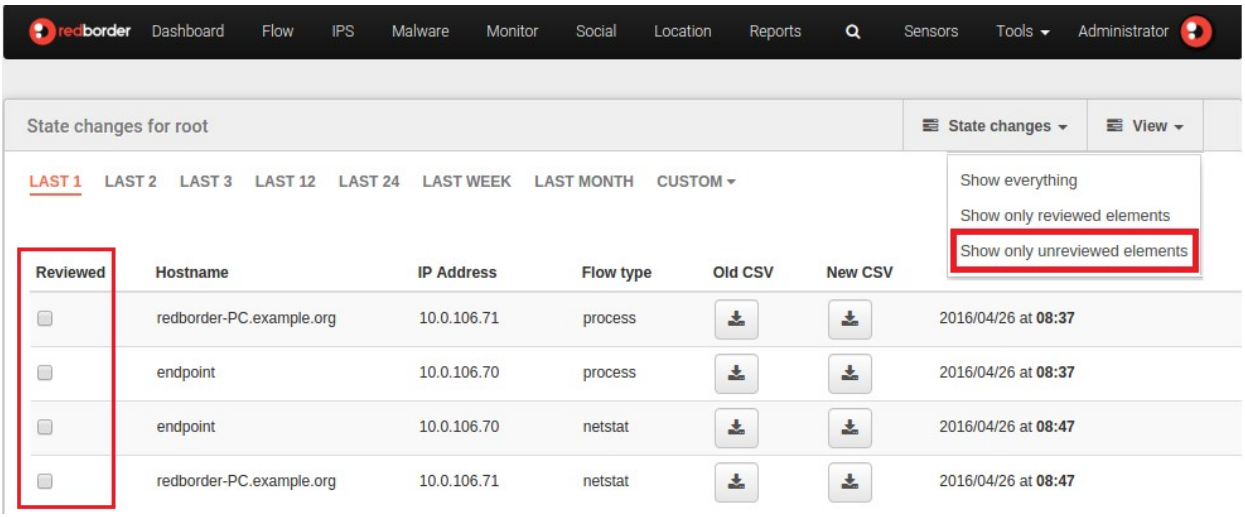

Figura 8.43 Vista de las entradas no revisadas

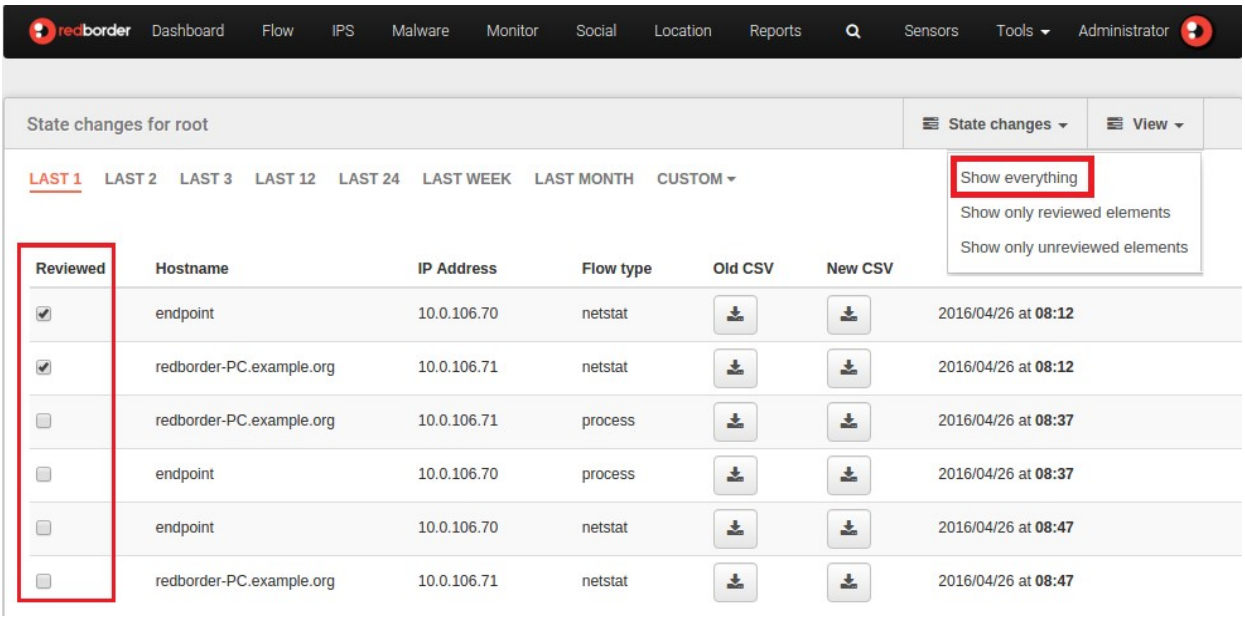

Figura 8.44 Vista de las entradas revisadas y no revisadas

9.DIAGRAMA DE GANTT

"No basta tener un buen ingenio, lo principal es aplicarlo bien"

- René Descartes -

Para ver el desarrollo de un proyecto a lo largo del tiempo es de utilidad la creación de un diagrama de Gantt. Con éste podemos visualizar todas las tareas que han sido necesarias para llevarlo a cabo y las relaciones entre ellas, a la vez que sus fechas de comienzo y duraciones. Esto es precisamente lo que veremos en este capítulo.

9.1 Tareas y subtareas

En este apartado explicaremos las diferentes tareas en las que se ha dividido el proyecto para su realización.

9.1.1 Integración GRR

9.1.1.1 Instalación agente GRR en EndPoint

Lo primero que hemos tenido que hacer es proceder al estudio de GRR y saber utilizarlo. Para ello instalamos el servidor en una VM Ubuntu Server 14.04. Seguido de ello nos descargamos el agente en un EndPoint. Vemos que todo ha salido correcto cuando en la web del servidor GRR aparece el equipo final.

9.1.1.2 Captura periódica de información del EndPoint

Ahora empezamos a enviar flows al sistema final y estamos atentos a la recepción de los resultados.

9.1.1.2.1 Descarga de snapshots del sistema del EndPoint

Para poder manipular los resultados obtenidos es necesario exportar los CSV al servidor GRR. Esto es de lo que nos encargamos aquí.

9.1.1.2.2 Envío de hunts a EndPoints

Al comienzo del proyecto se tenía en mente la utilización de los flows para obtener los análisis de Netstat y ListProcesses. Conforme el proyecto fue avanzando se vio que era más óptimo el envío de hunts, ya que de esta manera rbChanges sólo tenía que lanzar dos peticiones: una por cada flow.

9.1.1.3 Soporte para la identificación de cambios en el sistema

Es la hora de empezar a pensar la lógica para realizar un software capaz de detectar un cambio de estado en los EndPoints. Se parte con varias ideas hasta que se da con la que parece ser la más efectiva.

9.1.1.3.1 Comparación del estado actual con el anterior

Para establecer un cambio de estado se ha optado a comparar los CSV[55] de un equipo final en diferentes instantes de tiempo. El problema aquí surgió cuando cambiamos la táctica en el envío de peticiones a los EndPoints. Al lanzar hunts en lugar de flows los CSV van a incorporar ahora los resultados de todos los equipos. Para poder compararlos era necesario separarlos en diferentes ficheros. Este problema fue solventado en poco tiempo.

Una vez que tenemos los diferentes ficheros y les hacemos el hash podremos compararlos con sus antecesores y ver si son o no iguales.

9.1.1.3.2 Generación mensaje kafka al detectarse cambios, creación de la BBDD en Aerospike y envío de CSV a S3

Para hacer la comparación había que pensar en algún mecanismo con el que tener registrados los hash de los últimos ficheros que habían alertado un cambio de estado. Una de las opciones que se barajó fue dejarlo plasmado en un fichero de texto, pero rápidamente se descartó la idea porque es más engorroso obtener los parámetros que en una BBDD. Es por ello que se eligió hacerlo en Aerospike.

A parte de ello también era necesario avisar de los cambios de estado, por lo que se ha hecho uso de kafka.

Para poder visualizar los CSV desde la web era necesario subirlos a S3.

Todo ello es realizado en esta tarea.

9.1.1.4 Implementación de auditoría de acciones realizadas

Esta tarea es la encargada de registrar las acciones realizadas por GRR en un fichero.

9.1.1.5 Contención de equipos infectados

Para cortar toda comunicación entrante y saliente de los equipos afectados se contempló desde el primer momento hacer uso del Firewall de Windows.

Lo primero que fue necesario hacer para afrontar esta tarea era aprender a usar el FW. El alumno del proyecto no tenía conocimiento de los comandos necesarios para el manejo de éste en el SO Windows, por lo que un estudio básico fue fundamental.

9.1.1.5.1 Envío de mensaje vía GRR con comando con regla de FW de Windows

En esta tarea surgió el problema de cómo enviar las órdenes para modificar los FW a través de GRR. El alumno no tenía claro el funcionamiento de los python_hacks, ya que no había sido necesario el uso de ellos. Investigando ya se ideó el uso de éstos para programar (en python) una estrategia de ejecución de los comandos del FW en los EndPoints.

Tras esto fue sencillo el desarrollo de la tarea para producir la contención de equipos finales.

[55] De un mismo hunt.

9.1.2 Parada/Inicio servidor GRR en servidor

Esta tarea es la encargada de poder reiniciar, parar e iniciar el servicio GRR cuando veamos conveniente.

A continuación podemos ver todas estas tareas explicadas gráficamente en un diagrama de Gantt. Junto a ellas se encuentran el tiempo dedicado para cada una de ellas.

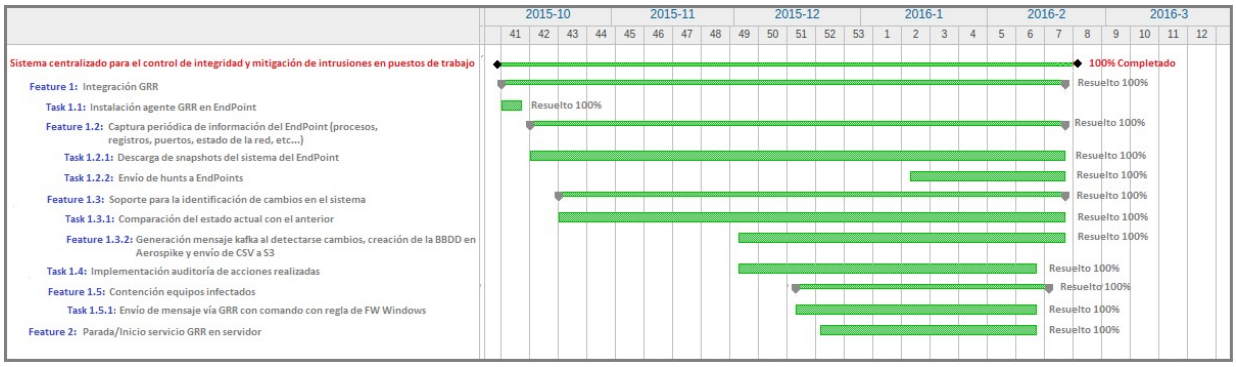

Figura 9.1 Diagrama de Gantt

10.PRESUPUESTO

"El valor del producto se halla en la producción"

- Albert Einstein -

En este apartado presentamos un breve presupuesto donde podemos ver las horas de trabajo necesarias para cada una de las partes de las que se compone el proyecto y su correspondiente valor económico.

Descripción	**Precio/Hora**	**Número de horas**	**Total (€)**
Estudio de GRR	35	180	6300
Definición de la estructura y funcionalidades de las diferentes partes del proyecto	35	60	2100
Implementación del proyecto	35	445	15575
Integración del proyecto en redBorder Malware	35	160	5600
Mantenimiento y mejora continua	35	140	4900
Total		**985 horas**	**34475 €**

Tabla 10.1 Presupuesto del proyecto

11. CONCLUSIONES

"Si no conozco una cosa, la investigaré"
- Louis Pasteur -

Desde la impartición de la asignatura de Seguridad en G.I.T.T por el tutor de este proyecto había sido de interés para el alumno la realización de un TFG relacionado con este temario. Gracias a la oportunidad brindada por redBorder ha sido posible llevarlo a cabo.

El malware está a la orden del día y hay que estar actualizados frente a las continuas amenazas que van apareciendo. Con este proyecto se pretende combatir la ciberdelincuencia de una manera diferente, a través de cambios de estado. Los resultados generados a la finalización del mismo han sido acogidos de manera satisfactoria por el cliente.

Las elaboraciones de nuevas herramientas no impiden ni menosprecian el uso de las ya existentes. Es por ello que podemos complementar este proyecto con un antivirus convencional. De este modo, al detectar en tiempo real una presencia malware en un EndPoint, podremos seguir el rastro dejado por el mismo. Analizando (p.e) el estado de la red, podríamos averiguar las conexiones (si corresponde) con las que se habrá comunicado el malware y, así, poder tomar las medidas oportunas para ser inmune a esas conexiones mediante la contención del firewall.

En definitiva, el proyecto ha cumplido con su objetivo: ser una fuente de lucha contra la ciberdelincuencia. A pesar de las dificultades encontradas, y del escaso manejo del alumno al comienzo del mismo de algunas de las herramientas usadas para llevarlo a cabo, ha sido bastante satisfactorio y gratificante haber llevado las riendas de este proyecto, a la vez que haber formado parte, por un tiempo, del equipo de redBorder.

11.1 Mejoras de cara al futuro

A continuación, enumeramos una serie de mejoras que se podrían implementar en este proyecto.

- Adición de nuevos flows para realizar los cambios de estado, no sólo Netstat y ListProcesses.
- Detección de cambios de estados acompañados de Indicadores de Compromiso (IOC).
- Dar la información necesaria al equipo web de redBorder para que se pudieran visualizar los EndPoints bloqueados en una lista. En estos momentos habría que ir uno por uno viendo si está aplicada o no la herramienta de contención.
- Para evitar falsos positivos, realizar un estudio intensivo de las columnas "estáticas" a dejar para establecer un cambio de estado. Actualmente son consideradas como estáticas las establecidas por el cliente.

ANEXO A: CÓDIGO DEL FICHERO PRINCIPAL DE RBCHANGES

Fichero con ruta en "~/grr_support.git/ grr_service/scripts/class_grr.py".

Código A.1 Contenido del fichero class_grr.py

```
import os
import sys
import hashlib
from Queue import Queue
import time
import threading
import shutil
import csv
from kafka import KafkaProducer
from kafka.common import KafkaError
from kafka import SimpleProducer, KafkaClient
import json
import aerospike
from aerospike.exception import *
import boto
from boto.s3.key import Key
from boto.s3.connection import S3Connection
import requests
from requests.auth import HTTPBasicAuth
import logging
import logging.handlers
import subprocess

CRON_HUNT = 1800

TMP = "/tmp/"
PATH_TMP_CSV = "/tmp/grrstates/temp_csv"
PATH_ENDPOINT_ID = "/tmp/grrstates/change_state/hunts"
PATH_LOG = "/var/log/grrstates/"
PATH = os.path.abspath(os.path.dirname(__file__))
LOG_CHANGES = "status_changes.log"
logger = init_logger(LOG_CHANGES, PATH_LOG, "changes")
```

```
#Function that checks the CSV has been created to it can be exported by rbChanges.
def check_csv(id_hunt, fich):
    server = 'http://localhost:8000'
    url = server + "/api/hunts/" + id_hunt + "/output-plugins"
    r = requests.get(url, auth=('admin', 'redborder'))
    if r.status_code == 200:
        res = str(r.content).replace(')]}\'', '')
        if "Results/CSVOutputPlugin/" + fich in res:
            return 1
    else:
        logger.error("GRR server returns a " + str(r.status_code) + "code.")

    return 0

#Function that export the CSV of the hunt throwed. At first, it checks if the
#CSV has been created with 'check_csv', insomuch as until all the EndPoints
#sends the results the CSV it isn't created. When the CSV exists it is exported
#and it creates new CSV, one for each EndPoint.
def get_csv(id_hunt, mode):
    if(mode == 'netstat'):
        fich = "ExportedNetworkConnection.csv"
    else:
        fich = "ExportedProcess.csv"

    time_sleep = 1
    while time_sleep != 0:
        csv_exists = check_csv(id_hunt, fich)
        if csv_exists == 1:
            time_sleep = 0
        else:
            time_sleep += 1
            time.sleep(60)
            if time_sleep == 8:
                logger.error("We do not get the " + fich +
                             " file. Are there any problem with GRR server?")
            if time_sleep > 12:
                logger.error("We do not get the " + fich +
                             " file. There are problems with GRR server")
                return -1

    #Command used to export the CSV to the path indicated in the variable
    #PATH_TMP_CSV
    cmd = ("grr_export file --path aff4:/hunts/" + str(id_hunt) +
           "/Results/CSVOutputPlugin/" + fich + " --output " + PATH_TMP_CSV)

    try:
        retcode = subprocess.call(cmd, shell=True)
        if retcode is not 0:
            logger.debug("get_csv: " + cmd + " returned " + str(retcode))
    except OSError as e:
        logger.error("get_csv: " + cmd + ". Execution failed:"  + str(e))
```

```
    #Here it wants to read the CSV to create new CSV, one for each id of the
    #EndPoints searched.
    try:
        path_csv_fich = (PATH_TMP_CSV + "/hunts/" + str(id_hunt) +
                         "/Results/CSVOutputPlugin/" + fich)
        csv_file = open(path_csv_fich, "rb")
        csv_arch =  csv_file.readlines()
        list_netstat = csv_arch[0]
        reader = csv.DictReader(csv_arch)
    except IOError:
        logger.error("We can not find the " + path_csv_fich + "file.")

    client = ""
    client_id = ""
    path = ''
    i = 2
    dir_ch_st = PATH_ENDPOINT_ID

    #It reads each line of the CSV. All the lines of the same EndPoint are written
    #in a new CSV and saved in a different path. This path will be created with
    #the id of the each EndPoint.
    for row in reader:
        if client != row['metadata.client_urn']:
            client = row['metadata.client_urn']
            client_id = client.replace('aff4:/', '')
            dir_csv = dir_ch_st + '/' + client_id
            dir_flow = dir_csv + '/' + mode
            if not os.path.exists(dir_csv):
                os.makedirs(dir_csv)
            if not os.path.exists(dir_flow):
                os.makedirs(dir_flow)
            path = dir_flow + '/' + mode + '_csv_' + client_id + '.csv'
            create_new_csv = open(path, "w")
            create_new_csv.write(list_netstat)
            create_new_csv.write(csv_arch[1])
            create_new_csv.close()
        else:
            create_new_csv = open(path, "a")
            create_new_csv.write(csv_arch[i])
            i += 1
            create_new_csv.close()

    csv_file.close()
    return 0

#Function that catchs the parameters from the file "parameters.json" and
#introduces them in an array. It depends if 'type_parameter' is s3, kafka
#or aerospike. It return the array.
def parameters(type_parameter):
    path_logs = os.path.join(PATH, "config", "parameters.json")
    with open(path_logs) as data:
        data_file = json.load(data)
    list_return = []
```

```
    if type_parameter == 's3':
        list_return.append(data_file["access_key"])
        list_return.append(data_file["secret_key"])
        list_return.append(data_file["bucket"])
    elif type_parameter == 'kafka':
        return data_file["kafka_broker"]
    elif type_parameter == 'aerospike':
        list_return.append(data_file["aerospike"])
        list_return.append(data_file["aerospike_user"])
        list_return.append(data_file["aerospike_password"])
        list_return.append(data_file["namespace"])
        list_return.append(data_file["table"])

    data.close()

    return list_return

#Function that make the hash of the individuals CSV.
def create_md5(filename):
    if os.path.exists(filename):
        flag = True
        md5_resume = None
        while flag==True or md5_resume is None:
            try:
                f = open(filename, "rb")
                content = f.read()
                md5_normal = hashlib.md5(content)
                md5_resume = md5_normal.hexdigest()
                f.close()
                flag = False
                return md5_resume
            except:
                logger.error('error make a md5')
                time.sleep(2)


#Function that sends the message kafka when it's produced a change of state.
#The arguments are the name of the old and the new CSV in s3, the tipe of the
#flow(Netstat or ListProcesses), id of the EndPoint, the md5 hash, and the
#kafka's producer respectively.
def send_kafka(s3_old, s3_new, mode, grr_identifier, md5_hash, producer_kafka):
    kafka_topic = "rb_changes"
    try:
        timestamp = int(time.time())
        message = ('{\"s3_old\":\"csv/' + s3_old +
                   '\",\"s3_new\":\"csv/' + s3_new +
                   '\",\"flow_type\":\"' + mode +
                   '\",\"md5\":\"' + md5_hash +
                   '\",\"endpoint_uuid\":\"' + grr_identifier +
                   '\",\"timestamp\":' + str(timestamp) + '}')
        producer_kafka.send(kafka_topic, message)
```

```
    except KafkaTimeoutError as kerror:
        logger.error("Failed to send this message to kafka: " + message)
        logger.error("Kafka: " + kerror.msg + " [" + kerro.code + "]")
    except Exception, e:
        s = str(e)
        logger.error("kafka: Exception: " + str(e))

#Function that creates the table in Aerospike.
def inicialize_aerospike():

    #It gets the parameters of aerospike from "parameters.json" in an array
    listar_parameters = parameters('aerospike')

    #Configure the client
    aerospike_host_port = str(listar_parameters[0])
    config = {
        'hosts': [(aerospike_host_port)]
    }

    # Create a client and connect it to the cluster
    try:
        aerospike_password = str(listar_parameters[1])
        aerospike_user = str(listar_parameters[2])
        client = aerospike.client(config).connect([aerospike_password,
                                                   aerospike_user])
        logger.debug("Successful connection to aerospike")
    except ClientError as e:
        string_logs = ("ERROR to connect to aerospike: \"" + str(e.msg) +
                                                   "\" [" + str(e.code) + "]")
        logger.error(string_logs)
        return 1

    #Create a key to write in aerospike
    aerospike_key = 'tariro'
    aerospike_namespace = str(listar_parameters[3])
    aerospike_table = str(listar_parameters[4])
    key = (aerospike_namespace, aerospike_table, str(aerospike_key))

    try:
    # Write an inicial record
        client.put(key, {
            'client_flow':'1',
            'client':'1',
            'md5':'1',
            's3_path':'1'
        })
        return 0
    except RecordError as e:
        string_logs = ("ERROR to write a record in aerospike: \"" +
                                    str(e.msg) + "\" [" + str(e.code) + "]")
        logger.error(string_logs)
        return 1

    # Close the connection to the Aerospike cluster
    #client.close()
```

```
#Function that introduce new entries en the aerospike's table.
def put_aerospike(s3_name, grr_identifier, md5_hash, mode, aerospike_key,
                  producer_kafka, status):

    # Catch parameters of aerospike from config/parameters.json
    listar_parameters = parameters('aerospike')
    # Configure the client
    aerospike_host_port = str(listar_parameters[0])
    config = {
        'hosts': [(aerospike_host_port)]
    }

    try:
        aerospike_password = str(listar_parameters[1])
        aerospike_user = str(listar_parameters[2])
        client = aerospike.client(config).connect([aerospike_password,
                                                   aerospike_user])
        logger.debug("AEROSPIKE: It's ready to receive new records")
    except ClientError as e:
        string_logs = ("AEROSPIKE: It have just failed the connection" +
                       + "to put new records: " + str(e))
        logger.error(string_logs)
        return 1

    # Create a client and connect it to the cluster
    try:
        aerospike_namespace = str(listar_parameters[3])
        aerospike_table = str(listar_parameters[4])
        key = (aerospike_namespace, aerospike_table, str(aerospike_key))
        bins = {
            'client_flow':str(aerospike_key),
            'client':str(grr_identifier),
            'md5':str(md5_hash),
            's3_path':str(s3_name)
        }

        if status == None or status == 'New':
            client.put(key, bins)
        elif status == 'exists':
            (primary_key, metadata, record) = client.get(key)
            old_s3_path = record['s3_path']
            client.put(key, bins)
            send_kafka(old_s3_path, s3_name, mode, grr_identifier, md5_hash,
                       producer_kafka)

        string_logs = 'AEROSPIKE: New event of client ' + grr_identifier +
                      ' type ' + mode + ' has been introduced succesfully'
        logger.debug(string_logs)
```

```
    #Exception launched if there are some error with the method 'put'
    except AerospikeError as e:
        string_logs = 'AEROSPIKE: Problem in event of client ' +
                      grr_identifier + ' type ' + mode +
                      ' to introduce in Aerospike: ' + str(e)
        logger.error(string_logs)
        return 1

    #Exception launched if there are some error with the method 'get'
    except RecordNotFound:
        string_logs = ("AEROSPIKE: Record not found: ", primary_key)
        logger.error(string_logs)

    #Exception launched if there are some error other than the above
    except Exception, e:
        logger.error("AEROSPIKE: Error: " + str(e))
        return 1

#Function that upload a CSV in s3. The first argument is the name os the csv and
#the second its rename in s3.
def s3_upload(csv_file, s3_name):
    list_parameters = parameters('s3')
    flag = True
    cont = 0

    while flag == True:
        try:
            #It connects to s3 ignoring the certificate
            conn = boto.connect_s3(aws_access_key_id = list_parameters[0],
                    aws_secret_access_key = list_parameters[1],
                    host = 's3.redborder.cluster',
                    validate_certs=False,
                    calling_format = boto.s3.connection.OrdinaryCallingFormat(),)

            #It gets the bucket where it will upload the CSV
            bucket = conn.get_bucket(list_parameters[2], validate=False)

            #It upload the CSV. The name will be the indicated as argument.
            k = Key(bucket)
            k.key = s3_name
            k.set_contents_from_filename(csv_file)
            flag = False
            logs_string = 'Upload ' + csv_file + ' to S3 with name ' + s3_name
            logger.debug(logs_string)
        except:
            if cont < 2:
                cont += 1
            else:
                flag = False
                logs_string = 'ERROR to upload ' + csv_file + ' to S3'
                logger.error(logs_string)
            continue
```

```
#Function that checks if it has produced a change of state. For that, rbChanges
#compares the hash of the old CSV with the new CSV.
def aerospike_request(aerospike_key, md5_hash):

    #It gets the parameters of aerospike from "parameters.json" in an array.
    listar_parameters = parameters('aerospike')

    # Configure the client
    config = {'hosts': [ (str(listar_parameters[0])) ]}

    # Create a client and connect it to the cluster
    retcode = None
    try:
        client = aerospike.client(config).connect([str(listar_parameters[1]),
                                                   str(listar_parameters[2])])
        key = (str(listar_parameters[3]), str(listar_parameters[4]),
               str(aerospike_key))

        #It obtains the hash from the database and compare it with the md5 of
        #the new CSV
        try:
            (primary_key, metadata, record) = client.get(key)
            if record != None:
                identifier_key = record['md5']
                if identifier_key == md5_hash:
                    retcode = 'OK'
                else:
                    retcode = 'exists'
            else:
                retcode = 'New'
            client.close()
            return retcode
        except:
            logger.error("Error in aerospike_request: client.get " + str(key))
            return retcode
        client.close()
    except:
        client.close()
        logger.error("Unexpected error: ", str(sys.exc_info()[0]))
        return retcode


#Function that creates the summary of the individuals CSV to gets only the
#static information.
def write_summary(fic_o, fic_d, mode):
    try:
        csv_arch = open(fic_o, "rb")
        reader = csv.DictReader(csv_arch)
        fic_d_file = open(fic_d, "w")

        #Depending of the flow's tipe, the columns of the CSV are differents.
        if mode == 'netstat':
            summary_netstat = csv.writer(fic_d_file)
```

```
            summary_netstat.writerow(['metadata.mac_address', 'family', 'type',
                                      'local_address.ip', 'local_adress.port',
                                      'remote_address.ip', 'remote_address.port',
                                      'pid'])
            #They are written the values of these columns in the summary.
            for row in reader:
                summary_netstat.writerow([row['metadata.mac_address'],
                                          row['family'], row['type'],
                                          row['local_address.ip'],
                                          row['local_address.port'],
                                          row['remote_address.ip'],
                                          row['remote_address.port'],
                                          row['pid']])

        elif mode == 'process':
            summary_process = csv.writer(fic_d_file)
            summary_process.writerow(['metadata.mac_address', 'pid', 'ppid',
                                      'name', 'exe', 'cmdline', 'username',
                                      'status', 'nice', 'cwd', 'num_threads',
                                      'user_cpu_time', 'cpu_percent', 'rss_size',
                                      'vms_size', 'memory_percent'])

            #They are written the values of these columns in the summary.
            for row in reader:
                summary_process.writerow([row['metadata.mac_address'], row['pid'],
                                          row['ppid'], row['name'], row['exe'],
                                          row['cmdline'], row['username'],
                                          row['status'], row['nice'],row['cwd'],
                                          row['num_threads'], row['user_cpu_time'],
                                          row['cpu_percent'], row['rss_size'],
                                          row['vms_size'], row['memory_percent']])
        else:
            logger.error("No 'netstat' or 'process' in file")

        csv_arch.close()
    except Exception as e:
        log = "Error in write_summary"
        logger.error(log)

#Function charged to make the summary of the CSV, their hash and compare them
#with the entries of the aerospike
def listar(queue_csv, producer_kafka):
    while True:
        try:
            #It gets the files of the queue. If it isn't anything, it is just
            #wait until arrive a new file
            csv_file = queue_csv.get()

            if os.path.exists(csv_file):
                if ("netstat" in csv_file) or ("process" in csv_file):
                    #It obtains the id of the EndPoint
                    list_word = csv_file.split("_")[-1]
                    grr_identifier = str(list_word).replace(".csv", "")
```

```
                mode = ''

                if("netstat" in csv_file):
                    mode = 'netstat'
                elif("process" in csv_file):
                    mode = 'process'

                #It creates a new CSV wich contains the summary of the
                #individual CSV
                file_temporal = grr_identifier + "_" + mode + ".csv"
                path_temporal = os.path.join(PATH, "temporal",  file_temporal)
                write_summary(csv_file, path_temporal, mode)

                #It makes the hash of the summary and queries the database
                #in aerospike
                md5_hash = create_md5(path_temporal)
                os.remove(path_temporal)
                aerospike_key = grr_identifier + '_' + mode
                s3_name = grr_identifier + '_' + md5_hash + ".csv"
                request_aerospike = aerospike_request(aerospike_key,
                                                      md5_hash)

                if request_aerospike == None or request_aerospike == 'New'
                                        or request_aerospike == 'exists':
                    s3_upload(csv_file, s3_name)
                    retcode = put_aerospike(s3_name, grr_identifier,
                                            md5_hash, mode, aerospike_key,
                                            producer_kafka, request_aerospike)
                os.remove(csv_file)
        except:
            logger.error("Error when get element in queue")
            continue


#Function charged to execute in the CLI Console of GRR the code located in the
#'file launch_process.py'. With it we throw the ListProcess hunt to the
#EndPoints.
def process_function(hunt_id_file_name):

    python_path = os.path.join(PATH, 'hunt_files/launch_process.py')
    if os.path.exists(python_path):
        #Command used to execute in a console python files in GRR
        cmd = "/usr/bin/grr_console --command_file " + python_path
        try:
            retcode = subprocess.call(cmd, shell=True)
            if retcode >= 1:
                if retcode == 1:
                    logger.debug("grr_console: launch_process.py: There are " +
                    + "not EndPoints")
                if retcode == 2:
                    logger.debug("grr_console: launch_process.py: They are " +
                    + "EndPoints with \"Outstanding\" status")
                return retcode
```

```
        except OSError as e:
            logger.error("grr_console_ netstat.py Execution failed:"  + str(e))

        #If the hunt has been throwed ok, the execution continue here. Now it
        #gets the id of the hunt written in the file passed as argument. This id
        #is written when the hunt is throwed. It needs it to get the CSV with
        #the function 'get_csv'.
        file_id_hunt = open(hunt_id_file_name, 'r')
        file_id_hunt_list = file_id_hunt.readlines()
        h = file_id_hunt_list[0].split("/")
        file_id_hunt.close()
        id_hunt = h[2].replace(">\n", "")
        ret = get_csv(id_hunt, 'process')
        if ret != 0:
           logger.error("process_function: We do not get the csv file.")
        return ret
    else:
        erorr_menssage = "process_function: " + python_path + " file does not exist"
        logger.error(erorr_menssage)
        return -1

#Function charged to execute in the CLI Console of GRR the code located in the
#'file launch_netstat.py'. With it we throw the Netstat hunt to the EndPoints.
def netstat_function(hunt_id_file_name):

    python_path = os.path.join(PATH, 'hunt_files/launch_netstat.py')
    if os.path.exists(python_path):
        #Command used to execute in a console python files in GRR
        cmd = "/usr/bin/grr_console --command_file " + python_path
        try:
            retcode = subprocess.call(cmd, shell=True)
            if retcode >= 1:
                if retcode == 1:
                    logger.debug("grr_console: launch_netstat.py: " +
                    + "There are not EndPoints")
                if retcode == 2:
                    logger.debug("grr_console: launch_netstat.py: They are " +
                    + "Endpoints with \"Outstanding\" status")
                return retcode

        except OSError as e:
            logger.error("grr_console_ netstat.py Execution failed:"  + str(e))

        #If the hunt has been throwed ok, the execution continue here. Now it
        #gets the id of the hunt written in the file passed as argument. This id
        #is written when the hunt is throwed. It needs it to get the CSV with
        #the function 'get_csv'.
        file_id_hunt = open(hunt_id_file_name, 'r')
        file_id_hunt_list = file_id_hunt.readlines()
        h = file_id_hunt_list[0].split("/")
        file_id_hunt.close()
        id_hunt = h[2].replace(">\n", "")
```

```
        ret = get_csv(id_hunt, 'netstat')
        if ret != 0:
            logger.error("Netstat_function: We do not get the csv file.")
        return ret
    else:
        erorr_menssage = "netstat_function: " + python_path + " file does not exist"
        logger.error(erorr_menssage)
        return -1

#Function used to connect to kafka. It returns the kafka's productor.
def kafka_init(kafka_server):
    producer_kafka = None
    connection = False
    while connection is False:
        try:
            producer_kafka = KafkaProducer(bootstrap_servers=kafka_server, retries=5)
            connection = True
        except:
            connection = False
            logger.error("ERROR to connect to kafka")
            logger.error("It will try to inicialize to kafka in 30 seconds")
            time.sleep(30)

    return producer_kafka

#Class that initializes kafka and aerospike. Also it puts in the queue the CSV
class csv_class(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.setDaemon(True)
    def run(self):
        path_grr = PATH_ENDPOINT_ID
        path_logs = os.path.join(PATH, "logs")
        path_temporal = os.path.join(PATH, "temporal")
        if not os.path.exists(path_logs):
            os.mkdir(path_logs)
        if not os.path.exists(path_temporal):
            os.mkdir(path_temporal)

        #It creates the queue
        q = Queue()

        #It gets the parameters of kafka from "parameters.json" in an array.
        kafka_server = parameters('kafka')

        #Then it try to connect to kafka. The function returns kafka's producer
        producer_kafka = kafka_init(kafka_server)
        logger.debug("Kafka init without errors.")
```

```
        #It creates and starts a thread. The thread will execute the function
        #'listar'. Its arguments are the queue and the kafka's producer.
        for num in range(1):
            h = threading.Thread(target=listar, args=(q,producer_kafka,))
            h.setDaemon(True)
            h.start()

        #It try to initialize to Aerospike.
        retcode = 1
        while retcode is 1:
            retcode = inicialize_aerospike()
            if retcode is 1:
                logger.error("It will try to connet to Aerospike in 30 seconds")
                time.sleep(30)

        while True:
            try:
                #It go into the directories of the EndPoints
                for element in os.listdir(path_grr):
                    if "C." in element:
                        #Inside of these directories there are the CSV with the
                        #results
                        path_grr_analysis = os.path.join(path_grr, element)
                        if os.path.exists(path_grr_analysis):
                            for root, dirs, files in os.walk(path_grr_analysis):
                                for file_new in files:
                                    #It catch the CSV and puts them in the queue
                                    file_csv = os.path.join(root, file_new)

                                    if os.path.isfile(file_csv) and
                                                    file_csv.endswith(".csv"):
                                        q.put(file_csv)
                time.sleep(60)
            except:
                logger.error("Error in csv class")
                time.sleep(60)
                continue


#Class charged to call the function that throws the Netstat hunt. Besides, it
#also creates the file 'hunt_netstat.txt'. This file is used to write inside the
#ID of the flow throwed.
class hunt_netstat(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.setDaemon(True)
    def run(self):
        hunt_id_file = PATH_ENDPOINT_ID + "/hunt_netstat.txt"
        dir_ch_st =  PATH_ENDPOINT_ID + '/'
        logger.debug("netstat: " + dir_ch_st)
        if not os.path.exists(dir_ch_st):
            os.makedirs(dir_ch_st)
```

```
        while True:
            ret = netstat_function(hunt_id_file)
            print "netstat sleep:" + str(CRON_HUNT) + " seconds."
            time.sleep(CRON_HUNT)


#Class charged to call the function that throws the ListProcesses hunt. Besides,
#it also creates the file 'hunt_netstat.txt'. This file is used to write inside
#the ID of the flow throwed.
class hunt_process(threading.Thread):
    def __init__(self):
        threading.Thread.__init__(self)
        self.setDaemon(True)
    def run(self):
        hunt_id_file = PATH_ENDPOINT_ID + "/hunt_process.txt"
        dir_ch_st = PATH_ENDPOINT_ID + '/'
        logger.debug("process: " + dir_ch_st)
        if not os.path.exists(dir_ch_st):
            os.makedirs(dir_ch_st)
        while True:
            ret = process_function(hunt_id_file)
            print "netstat sleep:" + str(CRON_HUNT) + " seconds."
            time.sleep(CRON_HUNT)

#Function that creates the log file
def init_logger(log_file, log_path, log_name):
    if not os.path.exists(log_path):
        os.mkdir(log_path)
    logger = logging.getLogger(log_name)
    logger.setLevel(logging.DEBUG)
    log_size = 2097152
    handler = logging.handlers.RotatingFileHandler(filename=log_path+log_file,
                                                   mode='a',
                                                   maxBytes=log_size,
                                                   backupCount=5)

    formatter = logging.Formatter(fmt='%(asctime)s - %(name)s - %(levelname)s' +
                                  + '- %(message)s',datefmt='%y-%m-%d %H:%M:%S')
    handler.setFormatter(formatter)
    logger.addHandler(handler)
    logger.debug('Creating the log file.')
    return logger


#The script starts here
if __name__ == '__main__':
    #calls to initiate classes
    csv_class().start()
    hunt_netstat().start()
    time.sleep(10)
    hunt_process().start()
    logger.debug("Class CSV initialize succesfully.")
```

```
try:
    while True:
        time.sleep(3)
except KeyboardInterrupt:
    logger.debug("Exit to CSV class")
    exit(0)
```

ANEXO B: CÓDIGO DEL FICHERO DE PETICIÓN DEL HUNT NETSTAT EN LA CLI CONSOLE

Fichero con ruta en "~/grr_support.git/grr_service/scripts/hunt_files/launch_netstat.py".

Código B.1 Contenido del fichero launch_netstat.py

```
# Script to automatize the hunts's sendings
# To throw it inside the CLI Console of GRR:
    # run -i launch_netstat.py
# To throw it out the CLI Console of GRR:
    # sudo /usr/bin/grr_console --command_file 'launch_netstat.py'

import os
from grr.lib import access_control
from grr.lib import flow_runner
from grr.lib.flows.general import file_finder
from grr.lib.hunts import implementation
from grr.lib.hunts import standard
from grr.lib.rdfvalues import flows as rdf_flows
from grr.lib import output_plugin

#Common arguments to all the hunts. They are used to configure the parameters
#of the hunt
runner_args = implementation.HuntRunnerArgs(
    hunt_name="GenericHunt",
    description="Running a Netstat.",
    client_rate=0,
    expiry_time="20m",
    regex_rules=[implementation.GRRHunt.MATCH_WINDOWS])

#Here it indicates the tipe of flow that will be throwed.
hunt_args = standard.GenericHuntArgs(
    output_plugins=[output_plugin.OutputPluginDescriptor(
        plugin_name="CSVOutputPlugin")],
    flow_runner_args=flow_runner.FlowRunnerArgs(
        flow_name="Netstat"))

#It throws the hunt to the EndPoints. The variable 'out_hunt' has the id of
#the hunt throwed.
out_hunt = implementation.GRRHunt.StartHunt(
    runner_args=runner_args,
    args=hunt_args,
    token=access_control.ACLToken(username="me",
        reason="Finding all the badness."))

#It changes the initial status of the hunt from 'paused' to 'started'
out_hunt.Run()
```

```
#It obtains the file 'hunt_netstat.txt' to write inside the id of the hunt
hunt_id_file = '/tmp/grrstates/change_state/hunts/hunt_netstat.txt'
f_id_h = open(hunt_id_file, 'w')
f_id_h.write(str(out_hunt) + '\n')
f_id_h.close()
h = str(out_hunt).split("/")
id_hunt = h[2].replace(">", "")

#parameters of the next loop
waiting = 0
finished_time = 0
started = []
completed = []
oustanding = []
onlineClients = []
countdown = 20

#Loop used to determined when all the EndPoints have completed the hunt
while ((countdown != 0) and (finished_time <= 3 and waiting <= 10)):
    results = aff4.FACTORY.Open("aff4:/hunts/" + id_hunt,
                                token=data_store.default_token)
    clients = results.GetClientsByStatus()

    #Check the number of online clients
    index = aff4.FACTORY.Create(client_index.MAIN_INDEX, aff4_type="ClientIndex",
                                mode="rw", object_exists=True,
                                token=data_store.default_token);

    #It obtains the list and the number of the all the clients that are
    #executing the hunt
    actualDate = datetime.datetime.now()
    actualYear = str(actualDate.year)
    actualMonth = str(actualDate.month)
    actualDay = str(actualDate.day)
    client_list = index.LookupClients(
                        ["start_date:" + actualYear
                        + "-" + actualMonth
                        + "-" + actualDay])
    onlineClients = len(client_list);

    #It catchs the array of the clients that are executing still the hunt
    #(outstanding), that have completed the hunt(completed) or that have started
    #it.
    oustanding = clients['OUTSTANDING']
    completed = clients['COMPLETED']
    started = clients['STARTED']

    #It checks that all the clients online have completed the hunt. It directly
    #finishes the loop.
    if((onlineClients == len(completed)) and (onlineClients != 0)):
        finished_time = 4
        continue
```

```
    #If there aren't any EndPoints online the parameter waiting increase one
    #unity. The loop don't finish yet.
    if (len(started) == 0 and len(oustanding) == 0 and len(completed) == 0):
        waiting += 1

    #If it has just completed the hunt in all the clients that they were online
    #the loop is finished
    if (len(started) > 0 and len(completed) > 0 and len(oustanding) == 0 and
        len(started) == len(completed)):
        finished_time += 1

    countdown -= 1
    time.sleep(60)


# with info about endpoints
if (len(completed) > 0):
    exit(0)

if (len(completed) == 0 and len(oustanding) ==0):
    exit(1)

if (len(oustanding) > 0):
    exit(2)
```

ANEXO C: CÓDIGO DEL FICHERO DE PETICIÓN DEL HUNT LISTPROCESSES EN LA CLI CONSOLE

Fichero con ruta en "~/grr_support.git/grr_service/scripts/hunt_files/launch_process.py".

Código C.1 Contenido del fichero launch_process.py

```
# Script to automatize the hunts's sendings
# To throw it inside the CLI Console of GRR:
    # run -i launch_process.py
# To throw it out the CLI Console of GRR:
    # sudo /usr/bin/grr_console --command_file 'launch_process.py'

import os
from grr.lib import access_control
from grr.lib import flow_runner
from grr.lib.flows.general import file_finder
from grr.lib.hunts import implementation
from grr.lib.hunts import standard
from grr.lib.rdfvalues import flows as rdf_flows
from grr.lib import output_plugin

#Common arguments to all the hunts. They are used to configure the parameters
#of the hunt
runner_args = implementation.HuntRunnerArgs(
    hunt_name="GenericHunt",
    description="Running a ListProcesses.",
    client_rate=0,
    expiry_time="20m",
    regex_rules=[implementation.GRRHunt.MATCH_WINDOWS])


#Here it indicates the tipe of flow that will be throwed.
hunt_args = standard.GenericHuntArgs(
    output_plugins=[output_plugin.OutputPluginDescriptor(
        plugin_name="CSVOutputPlugin")],
    flow_runner_args=flow_runner.FlowRunnerArgs(
        flow_name="ListProcesses"))

#It throws the hunt to the EndPoints. The variable 'out_hunt' has the id of
#the hunt throwed.
out_hunt = implementation.GRRHunt.StartHunt(
    runner_args=runner_args,
    args=hunt_args,
    token=access_control.ACLToken(username="me",
        reason="Finding all the badness."))

#It changes the initial status of the hunt from 'paused' to 'started'
out_hunt.Run()
```

```
#It obtains the file 'hunt_process.txt' to write inside the id of the hunt. It
#is necessary to export the CSV later.
hunt_id_file = '/tmp/grrstates/change_state/hunts/hunt_process.txt'
f_id_h = open(hunt_id_file, 'w')
f_id_h.write(str(out_hunt) + '\n')
f_id_h.close()
h = str(out_hunt).split("/")
id_hunt = h[2].replace(">", "")

#parameters of the next loop
waiting = 0
finished_time = 0
started = []
completed = []
oustanding = []
onlineClients = []
countdown = 20

#Loop used to determined when all the EndPoints have completed the hunt
while ((countdown != 0) and (finished_time <= 3 and waiting <= 10)):
    results = aff4.FACTORY.Open("aff4:/hunts/" + id_hunt,
                                token=data_store.default_token)
    clients = results.GetClientsByStatus()

    #Check the number of online clients
    index = aff4.FACTORY.Create(client_index.MAIN_INDEX, aff4_type="ClientIndex",
                                mode="rw", object_exists=True,
                                token=data_store.default_token);


    #It obtains the list and the number of the all the clients that are
    #executing the hunt
    actualDate = datetime.datetime.now()
    actualYear = str(actualDate.year)
    actualMonth = str(actualDate.month)
    actualDay = str(actualDate.day)
    client_list = index.LookupClients(
                        ["start_date:" + actualYear
                        + "-" + actualMonth
                        + "-" + actualDay])
    onlineClients = len(client_list);


    #It catchs the array of the clients that are executing still the hunt
    #(outstanding), that have completed the hunt(completed) or that have started
    #it.
    oustanding = clients['OUTSTANDING']
    completed = clients['COMPLETED']
    started = clients['STARTED']

    #It checks that all the clients online have completed the hunt. It directly
    #finishes the loop.
    if((onlineClients == len(completed)) and (onlineClients != 0)):
        finished_time = 4
        continue
```

```
    #If there aren't any EndPoints online the parameter waiting increase one
    #unity. The loop don't finish yet.
    if (len(started) == 0 and len(oustanding) == 0 and len(completed) == 0):
        waiting += 1

    #If it has just completed the hunt in all the clients that they were online
    #the loop is finished
    if (len(started) > 0 and len(completed) > 0 and len(oustanding) == 0 and
        len(started) == len(completed)):
        finished_time += 1

    countdown -= 1
    time.sleep(60)


# with info about endpoints
if (len(completed) > 0):
    exit(0)

if (len(completed) == 0 and len(oustanding) ==0):
    exit(1)

if (len(oustanding) > 0):
    exit(2)
```

ANEXO D: EJEMPLO DEL FICHERO DE CONFIGURACIÓN DE RBCHANGES

Fichero con ruta en "~/grr_support.git/ grr_service/scripts/config/parameters.json".

Código D.1 Contenido del fichero parameters.json

```
{"access_key":"V7IUM8EHNU_VF6B9GIG7","secret_key"
  :"2XmRE9JD86_yYS2g3DtxgN_Ys2w3_Vveg7hnpQ==","bucket":"malware/csv/"
  ,"kafka_broker":"rbmivlfwged6.redborder.cluster:9092","aerospike":"managerGRR
  .redborder.cluster:3000","aerospike_password":"redborder","aerospike_user"
  :"root","namespace":"malware","table":"grr"}
```

De él podemos sacar las siguientes correspondencias:

```
access_key : V7IUM8EHNU_VF6B9GIG7
secret_key : 2XmRE9JD86_yYS2g3DtxgN_Ys2w3_Vveg7hnpQ==
bucket : malware/csv/
kafka_broker : rbmivlfwged6.redborder.cluster:9092
aerospike : managerGRR.redborder.cluster:3000
aerospike_password : redborder
aerospike_user : root
namespace : malware
table : grr
```

ANEXO E: CÓDIGO DEL PYTHON_HACK PARA BLOQUEAR LA CONEXIÓN DE UN ENDPOINT EN RBCONTENTION

Fichero con ruta en "~/grr_support/python_hacks/firewall/block_endpoint.py".

Código E.1 Contenido del fichero block_endpoint.py

```
import os
import sys

#list of allowed ips
parameter_ips = py_args['allowed_ips']
ips = ""

#array of alloweds ips
for i in parameter_ips:
        if i != parameter_ips[-1]:
                ips += str(i) + ','
        else:
                ips += str(i)
magic_return_str = "Blocked IPs :" + str(ips)

#it activates the firewall
os.system('netsh advfirewall set allprofiles state on')

#it blocks all connections inbounds and outbounds
os.system('netsh advfirewall set allprofiles firewallpolicy blockinbound,' +
        + 'blockoutbound')

#it allows an array of ips
os.system('netsh advfirewall firewall add rule name=\"allow ips inbound\" ' +
        + 'dir=in action=allow protocol=any remoteip=' + ips)
os.system('netsh advfirewall firewall add rule name=\"allow ips outbound\" ' +
        + 'dir=out action=allow protocol=any remoteip=' + ips)

magic_return_str += "- Finished"
```

ANEXO F: CÓDIGO DEL PYTHON_HACK PARA DESBLOQUEAR LA CONEXIÓN DE UN ENDPOINT EN RBCONTENTION

Fichero con ruta en "~/grr_support/python_hacks/firewall/unblock_endpoint.py".

Código F.1 Contenido del fichero unblock_endpoint.py

```
import os
import sys


#it activates the firewall
os.system('netsh advfirewall set allprofiles state on')

#it blocks inbount connections and allows outbound connections
os.system('netsh advfirewall set allprofiles firewallpolicy blockinbound, ' +
          + 'allowoutbound')

#it deletes the rules applieds by block_endpoint.py
os.system('netsh advfirewall firewall delete rule name=\"allow ips inbound\"')
os.system('netsh advfirewall firewall delete rule name=\"allow ips outbound\"')

magic_return_str = "finished"
```

ANEXO G: CÓDIGO DEL FICHERO ENCARGADO DE LA REALIZACIÓN DE RBAUDIT

Fichero con ruta en “~/grr_support/grr_summary/py_summary/class_summary.py".

Código G.1 Contenido del fichero class_summary.py

```
import sys
import os
from Queue import Queue
import time
import threading

#Function used to find out the size of a file passed as argument
def get_size_file(filename):
  var = None
  try:

    #If the file exists it's calculated its size
    if os.path.exists(filename):
      tamanio = os.path.getsize(filename)
      return tamanio

    #If the file don't exists...
    else:
      return var
  except:
    return var

#Function used to write the logs of the programm
def logs_function(cadena):
  path = os.path.abspath(os.path.dirname(__file__))
  path_logs = os.path.join(path, "logs", "grr_logs.txt")
  try:
      #coge el tiempo actual del sistema y lo pone en hora normal
      time_now = time.asctime(time.localtime(time.time()))

      if not os.path.exists(path + '/logs'):
        os.makedirs(path + '/logs')

      #If the follow path don't exits it's created and it's written with the
      #time and the argument's string
      if not os.path.exists(path_logs):
        f = open(path_logs, 'w')
        f.write(str(time_now + ' ' + cadena + '\n'))
        f.close()
```

```
      else:
        #If the path exists it finds out its size
        tamanio = get_size_file(path_logs)

        if tamanio != "" and tamanio != None:
          if tamanio < 262144:
            #the file is written with the time and the argument's string
            fd = open(path_logs, 'a')
            fd.write(str(time_now) + ' ' + cadena + '\n')
            fd.close()
          else:
            #the original file is read and the last 50 lines are caught. That
            #file is overwritten with those lines
            fh = open(path_logs, 'r').readlines()
            lines = fh[-50:]
            fx = open(path_logs, 'w')
            fx.write(lines)
            #Besides it is added the time
            fx.write(str(time_now) + ' ' + cadena + '\n')
            fx.close()
  except:
    pass

#Class used to execute the principal function of the script
class summary_class(threading.Thread):
        def __init__(self):
            threading.Thread.__init__(self)
            self.setDaemon(True)
        def run(self):
        logs_function("Class Summary has just initialized succesfully")
        path = os.path.abspath(os.path.dirname(__file__))

        #File used to execute the code in the CLI Console
        python_path = os.path.join(path, 'py_summary/see_summary.py')

        #Command used to execute python files in a console in GRR
        os.system('/usr/bin/grr_console --command_file \'' + python_path +'\'')
        logs_function("It have just obtain GRR's behavior")


#The script starts here
try:
    t = summary_class()
    t.start()
    t.join()

except KeyboardInterrupt:
        new_string = "Exit from Summary class"
        logs_function(new_string)
        exit(0)
```

ANEXO H: CÓDIGO DEL FICHERO ENCARGADO DE EJECUTARSE EN LA CLI CONSOLE PARA LA REALIZACIÓN DE RBAUDIT

Fichero con ruta en "~/grr_support/grr_summary/py_summary/see_summary.py".

Código H.1 Contenido del fichero see_summary.py

```
#Script used to execute the follow commands in the CLI Console

import os
import sys

#File used to write it the auditory of GRR
path_file_audit = 'audits_grr.txt'

file_audit = open(path_file_audit, 'w')

audits = [ ]

#Command used to obtain the auditory's logs of GRR in the CLI Console
logs_urn = aff4.FACTORY.Open("aff4:/audit/logs", token=data_store.default_token)

#Loop used to add in an array all the files of the auditory
for child in logs_urn.ListChildren():
    audits.append(str(child))

#It gets the last file of the array
last_audit = audits[-1]

#It divides the logs in a list
log = aff4.FACTORY.Open(last_audit, token=data_store.default_token)
entries = list(log)

#Loop used to write the logs in the txt file
for x in entries:
    file_audit.write(str(x) + '\n')

file_audit.close()
```

ANEXO I : CÓDIGO DEL SCRIPT ENCARGADO DE PARAR Y REINICIAR EL SERVICIO GRR EN RBST

Código H.1 Contenido del script shell_helpers.sh

Este fichero ha sido creado por los desarrolladores de GRR. Sin embargo, para adaptarlo a las necesidades de este proyecto, el alumno lo ha modificado.

```
#!/bin/bash
# Some basic functions to help with enabling and disabling GRR services.

GRR_SERVICES="grr-http-server grr-ui grr-worker"

alias grr_stop_all='stop_services "$GRR_SERVICES"'
alias grr_start_all='start_services "$GRR_SERVICES"'
alias grr_enable_all='enable_services "$GRR_SERVICES"'
alias grr_restart_all='grr_stop_all; grr_start_all'

#function used to stop the services of GRR
function stop_services()
{
  local SERVICES;
  SERVICES=$1;
  echo "SERVICES: [$SERVICES]"
  for SERVICE in ${SERVICES}
  do
    if [ -x $(which service) ]; then
      sudo service ${SERVICE} status | grep "running"
        IS_RUNNING=$?
        if [ $IS_RUNNING = 0 ]; then
          echo "Stopping ${SERVICE}"
          sudo service ${SERVICE} stop;
        fi
    else
      echo "Systems that don't use 'service' not supported"
      exit 1
    fi
  done
}

#function used to start the services of GRR
function start_services()
{
  local SERVICES;
  SERVICES=$1;
  for SERVICE in ${SERVICES}
  do
    if [ -x $(which initctl) ]; then
```

```
      sudo service ${SERVICE} status | grep "stop"
        IS_RUNNING=$?
        if [ $IS_RUNNING = 0 ]; then
          sudo service ${SERVICE} start;
        fi
      else
        echo "Systems that don't use 'service' not supported"
        exit 1
      fi
    done
}

#function used to enable the services of GRR
function enable_services()
{
  local SERVICES;
  SERVICES=$1;
  for SERVICE in ${SERVICES}
  do
    SERVICE_DEFAULT=/etc/default/${SERVICE}
    sed -i 's/START=\"no\"/START=\"yes\"/' ${SERVICE_DEFAULT}:
    initctl status ${SERVICE} | grep "running"
    IS_RUNNING=$?
    if [ $IS_RUNNING = 0 ]; then
      service ${SERVICE} stop
    fi
    service ${SERVICE} start
  done
}

#this script will execute a function or another depending the argument passed
#with its called
case $1 in
    "grr_stop_all")
        stop_services "$GRR_SERVICES"
        ;;
    "grr_start_all")
        start_services "$GRR_SERVICES"
        ;;
    "grr_enable_all")
        enable_services "$GRR_SERVICES"
        ;;
    "grr_restart_all")
        stop_services "$GRR_SERVICES"
        start_services "$GRR_SERVICES"
        ;;
    *)
        show_usage
        ;;
esac
```

ANEXO J : FLOWS DE GRR

Los flows, como ya sabemos, son las diferentes peticiones realizadas desde el servidor GRR hacia un cliente para poder obtener información específica acerca de éste y, así, poder proceder a analizarlo.

Aunque no sea objeto de este proyecto el conocimiento de los diferentes flows existentes en GRR, lo dejamos como curiosidad.

A continuación, se presenta gran parte del abanico de flows que podemos lanzar:

Tabla J.1 Flows de GRR

CATEGORÍA	FLOWS	EXPLICACIÓN
BROWSER	**ChromeHistory**	Obtiene el historial del navegador de Chrome
	FirefoxHistory	Obtiene el historial del navegador de Firefox
CHECKS	**CheckRunner**	Flujo que corre comprobaciones en un host y devuelve el resultado de dicho análisis.
COLLECTORS	**ArtifactCollectorFlow**	Flujos para coleccionar los resultados de los diferentes artefactos forenses. Éstos son definidos en YAML. Son muy similares a los IOCs pero, al contrario que éstos, describen únicamente datos, mientras que los IOCs son descritos de una manera más lógica, utilizando los operadores lógicos 'AND' y 'OR' para determinar la condición

		que ha de cumplirse.
FILETYPES	**PlistValueFilter**	Obtiene los valores de una lista de propiedades basada en un contexto y un filtrado, devolviendo el resultado dado. **Ejemplo:** plist = { 'values': [13, 14, 15] 'items': [{'name': 'John', 'age': 33, 'children': ['John', 'Phil'], }, {'name': 'Mike', 'age': 24, 'children': [], },], } Si llamamos a PlistValueFilter con el contexto "items" y como filtrado "age > 25" nos devolverá {'name': 'John', 'age': 33, 'children': ['John', 'Phil']}.
FILESYSTEM	**DiskVolumeInfo**	Obtiene la información acerca del volumen de un disco del equipo cliente que le indiquemos mediante una ruta
	FileFinder	Este flujo busca archivos que coinciden con una serie de criterios y actúa sobre ellos. Acciones a realizar:{STATS, HASH, DOWNLOAD}
	FindFiles	Encuentra ficheros/directorios en un cliente. Para ello le pasamos una ruta y el nombre del fichero/directorio, y lo busca A PARTIR de esa ruta hacia abajo. Tiene

		como incoveniente su ineficiencia para un número grande de ficheros
	FingerprintFile	Obtiene el hash de un fichero especificado por la ruta correspondiente.
	GetFile	Mecanismo eficiente de transferencia de archivos. También puede recibir contenido de los archivos de un dispositivo cuando tiene un tamaño de 0 bytes cuando el campo read_length es especificado
	GetMBR	Flujo con el que obtenemos el MBR. El MBR es el primer sector del disco duro del ordenador que le dice al PC cómo cargar el sistema operativo y cómo el disco duro está particionado. Sólo va a haber una sóla sección MBR. Hay virus que atacan al MBR evitando que el equipo arranque.
	ListDirectory	Nos devuelve la lista los ficheros de un directorio, proporcionando anteriormente su ruta.
	ListVolumeShadowCopies	Shadow Copy es la Tecnología incluida en Microsoft que permite tomar copias de

		seguridad instantáneas tanto manuales como automáticas de archivos , incluso cuando están en uso. Se implementa como un servicio de Windows. Lo que hace este flujo es listar todo ello.
	SendFile	Flujo que envia fichero a un receptor remoto. Para ello se especifica IP y un sistema de claves como sistema de seguridad de transmisión/recepción de los archivos.
MEMORY	**AnalyzeClientMemory**	Analiza memoria RAM del cliente usando rekall (marco de análisis de memoria más completa). Es la única plataforma de análisis de memoria específicamente diseñada para ejecutarse en la misma plataforma en la que se está analizando.
	DownloadMemoryImage	Copia la imagen de memoria de un disco local y luego recupera el archivo
	ListVADBinaries	Obtiene la lista de todos los binarios que corren desde Rekall y las coincidencias con él.
	LoadMemoryDriver	Carga un controlador de memoria en el cliente. El flujo AnalyzeClientMemory hará esto si lo llamamos.
	MemoryCollector	Flujo que realiza un análisis de la memoria del cliente estableciendo previamente una serie de

		condiciones en el envío de la petición y aplicando la acción elegida (p.e download) si se cumple el resultado de la consulta con los requisitos expuestos. Lo que nos devuelve este flujo son precisamente las coincidencias resultantes.
	ScanMemory	Flujo que obtiene parte de la memoria del cliente basándose en una serie de firmas.
	UnloadMemoryDriver	Descarga un controlador de memoria en el cliente.
MISC	**TakeScreenshot**	Toma una captura de pantalla del sistema cliente en funcionamiento. Restricción encontrada: Sólo soportado en sistemas operativos OS X.
NETWORK	**Netstat**	Flujo que muestra un listado de las conexiones activas de un equipo clietne, tanto entrantes como salientes.
PROCESSES	**ListProcesses**	Informa de los procesos corriendo en el equipo cliente.
REGISTRY	**CollectRunKeyBinaries**	Nos devuelve los binarios usados por las llaves Run y RunOnce del sistema.

	GetMRU	Flujo que devuelve una colección de la lista de ficheros más recientes usados por los usuarios del equipo cliente.
	RegistryFinder	Flujo que realiza una búsqueda de los registros del sistema dados por cierto criterio.
TIMELINE	**MACTimes**	Flujo que proporciona las marcas de tiempo de la última modificación (mtime) o la última vez por escrito, el acceso (atime) o el cambio (ctime) de un determinado archivo dado mediante su ruta.

ANEXO K : VISTAS DEL PROYECTO EN LA WEB DE REDBORDER

En este anexo se explicarán las diferentes vistas web del proyecto realizadas por el equipo de redBorder.

Aclaremos, antes de nada, que el alumno no ha realizado estas interfaces gráficas para el TFG, sino el equipo web de redBorder. El alumno ha contribuido en ello adaptando parte de la implementación del proyecto para poder visualizar en la web las distintas partes del mismo. No obstante, se ha considerado oportuno explicarlas ya que se van a hacer referencias a ellas en el capítulo 8.

Comencemos.

VISTA DE ENDPOINTS

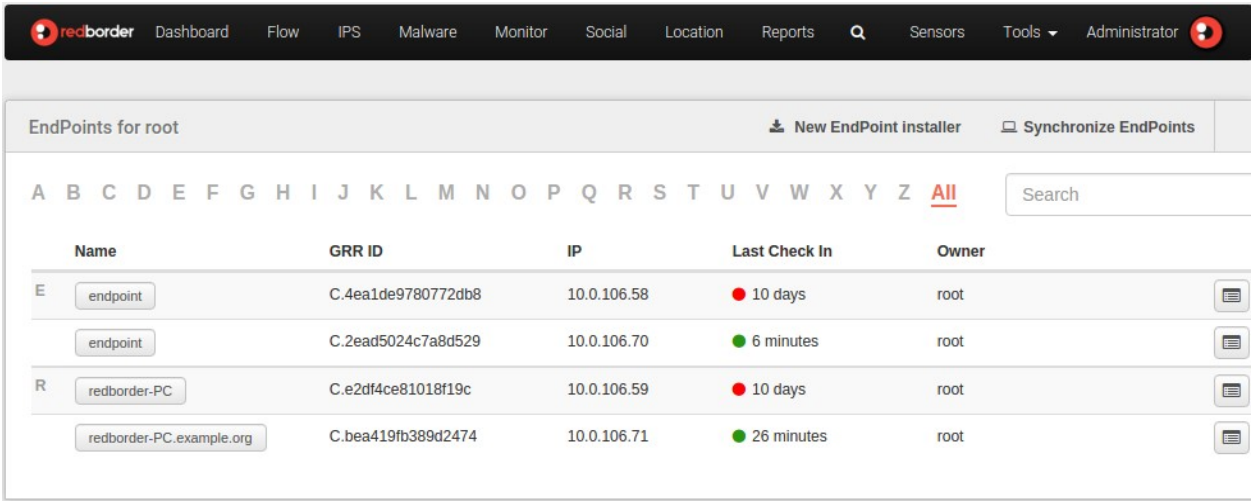

Figura K.1 Vista de EndPoints

Esta vista, ubicada en /endpoints, muestra la lista de todos los equipos finales registrados con GRR. En ella podemos el nombre que tiene cada equipo, su identificador (proporcionado por GRR al registrarse), su ip, la última vez que se detectó que estaba encendido y el usuario de GRR que se ha encargado de registrarlos.

También tenemos la posibilidad de buscar los sistemas finales alfabéticamente por su nombre o realizar una búsqueda explícita.

Pulsando sobre el botón ubicado más a la derecha podremos ir hasta la vista de estado de un EndPoint (explicado a continuación).

Por último, desde aquí también será desde donde nos descargaremos el instalador en los EndPoints para registrarnos (New EndPoint Installer) y donde podremos sincronizarlos para que aparezcan en la web tras su registro (Synchronize EndPoints).

VISTA DE ESTADO DE UN ENDPOINT

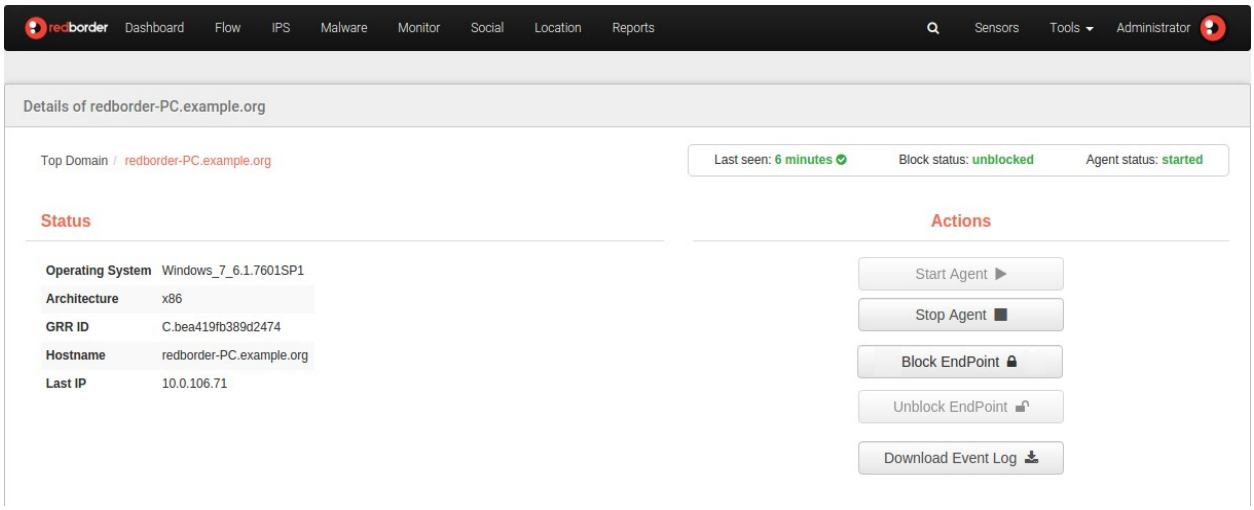

Figura K.2 Vista de estado de un EndPoint

Esta vista, ubicada en /endpoints/status, nos da la información particular de un equipo final, el que hayamos elegido.

A la izquierda nos muestra el SO del EndPoint, su arquitectura, identificador GRR, nombre e IP.

En la parte derecha tenemos la posibilidad de bloquear o desbloquear el EndPoint usando la contención del firewall de Windows. Para ello existen los botones *Block EndPoint* y *Unblock EndPoint*. Igualmente, podemos ver en qué estado de bloqueo se encuentra en estos momentos el equipo final (Block status): en el caso de ser *unblocked* significa que no está aplicada la contención, y si es *blocked* es porque sí lo está.

Los demás botones que aparecen no tienen que ver con este proyecto, es por ello que los obviamos.

VISTA DE CONFIGURACIÓN DE MALWARE

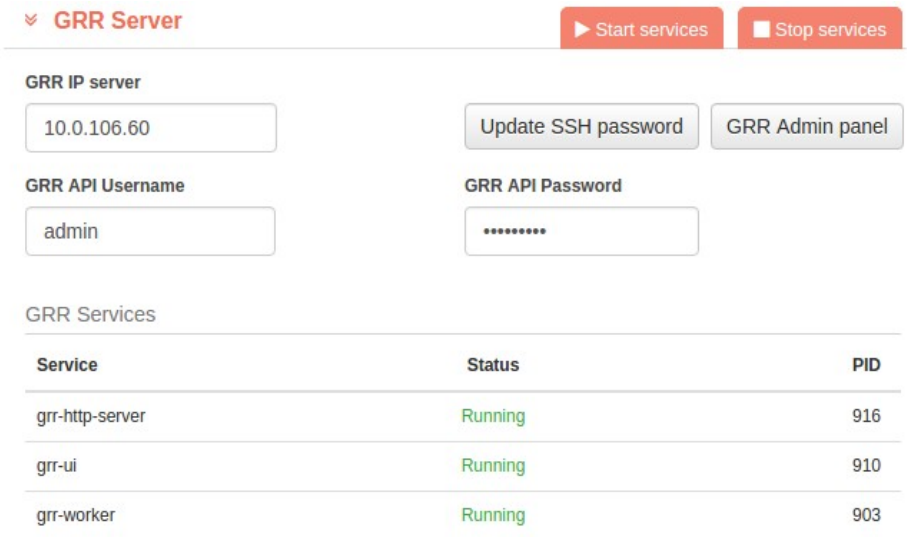

Figura K.3 Vista de configuración de Malware

Esta vista, ubicada en /malware/settings, se encarga de iniciar y parar el servicio GRR, a la vez que configurar los parámetros necesarios para conectarnos con él.

Con los botones *Start services* y *Stop services* se inicia y para respectivamente el servidor GRR. Si pulsamos este último, automáticamente los servicios de GRR que aparecen más abajo tendrán el estado de *Stopped* en lugar de *Running*, desapareciendo, además, el PID de cada uno de ellos.

Por último, podremos acceder a la web de GRR pulsando sobre *GRR Admin Panel*. Desde allí podremos monitorizar gráficamente el envío de flows y hunts, a la vez que ver sus resultados y tener un control de los EndPoints registrados. Para ello en la casilla de IP del servidor GRR se debe introducir el nodo que tiene la VM de GRR. Además, deberemos incluir las credenciales dadas al descargar GRR en la VM para acceder a la interfaz web de éste (*GRR API Username* y *GRR API Password*).

VISTA DE CAMBIOS DE ESTADO

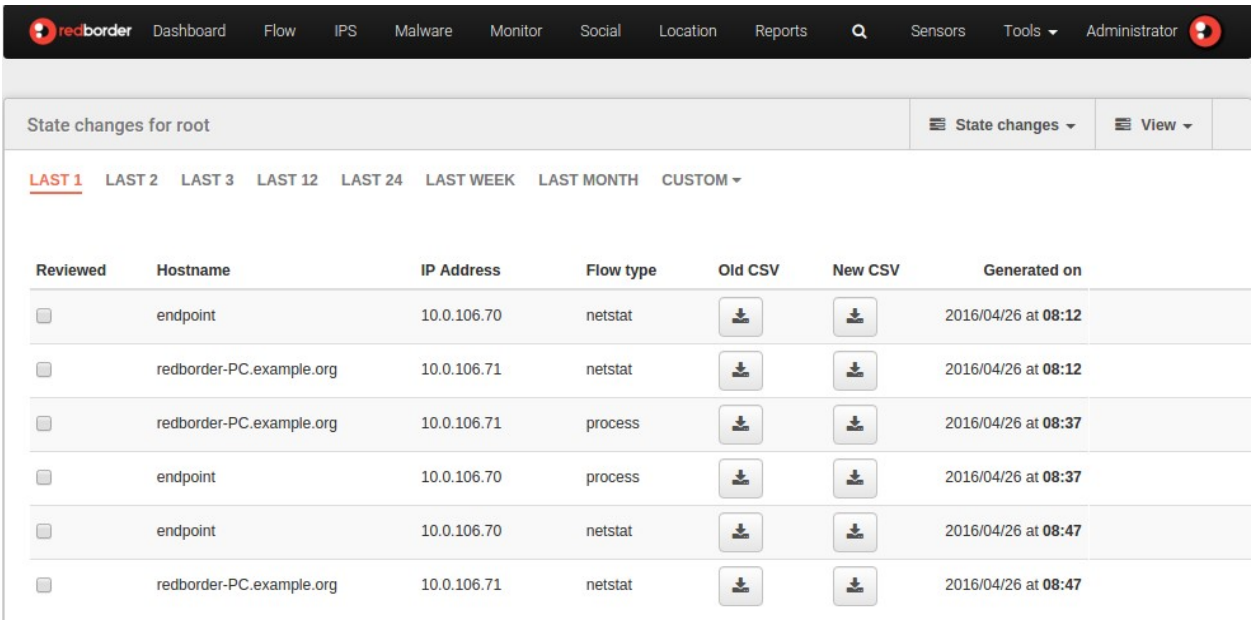

Figura K.4 Vista de cambios de estado

Esta vista, ubicada en /endpoints/state_changes, es desde donde podremos ver si se ha producido un cambio de estado. En ese caso podremos descargarnos el nuevo CSV y su antecesor para poder analizar los cambios producidos.

Para empezar, podemos ver que podemos cambiar la vista hasta por las últimas 24 horas y por la última semana y mes. Dependiendo lo que elijamos nos aparecerán unos cambios de estados u otros (o incluso ninguno si no se llegara a producir).

A continuación, tendremos la posibilidad de señalar las entradas revisadas para llevar un control de lo analizado gracias a la opción *Reviewed*. También podremos identificar los EndPoints a través de su nombre e IP. Por último, tenemos información acerca del hunt: de qué tipo de flow se trata, los dos botones para la descarga del antiguo y nuevo CSV y la fecha y hora en la que se ha producido la entrada.

Para terminar, tenemos arriba una pestaña desplegable con el nombre de *View*. Ésta sirve para filtrar

las entradas de los cambios de estado. Podremos ver sólo las marcadas en casilla *Review*, las no marcadas o todas. A continuación, podemos ver este menú desplegable.

Figura K.5 Tipos de filtrado de cambios de estado

REFERENCIAS

[1] GRR, «Web de Github » 2015. [En línea]. Available: https://github.com/google/grr

[2]GRR, « Comienzos rápidos de instalación » 2016. [En línea]. Available: https://github.com/google/grr-doc/blob/master/quickstart.adoc

[3] redBorder, «Web de redBorder» 2016. [En línea]. Available: https://redborder.com/.

[4] Kafka, «Web de Kafka » 2016. [En línea]. http://kafka.apache.org/

[5] Aerospike, «Web de Aerospike » 2016. [En línea]. http://kafka.apache.org/http://www.aerospike.com/

[6] S3, «Web de Amazon S3 » 2016. [En línea]. https://aws.amazon.com/es/s3/

[7] Python, «Web de Python » 2016. [En línea]. https://www.python.org/

[8] GRR, «Web de Github » 2015. [En línea]. Available: https://github.com/google/grr

[9] M.I. Cohen, D. Bilby, G. Caronni. (2011). Distributed forensics and incident response in the enterprise. Digital Investigation, 8, 101-110.

[10] Moser, Andreas, and Michael I. Cohen. (2013). Hunting in the enterprise: Forensic triage and incident response. Digital Investigation, 10(2), 89-98.

[11] M.I. Cohen, D. Bilby, G. Caronni. (2011). Distributed forensics and incident response in the enterprise. Digital Investigation, 8, 101-110.

[12] M.I. Cohen, D. Bilby, G. Caronni. (2011). Distributed forensics and incident response in the enterprise. Digital Investigation, 8, 101-110.

[13] M.I. Cohen, D. Bilby, G. Caronni. (2011). Distributed forensics and incident response in the enterprise. Digital Investigation, 8, 101-110.

ÍNDICE DE COMANDOS

ÍNDICE DE CÓDIGOS